Cooperating Languages - Core Language

Hesam Samimi (advised by Alan Borning)

This material is based upon work supported in part by the National Science Foundation under Grant No. 0639876. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

VPRI Memo M-2012-004

Viewpoints Research Institute, 1209 Grand Central Avenue, Glendale, CA 91201 t: (818) 332-3001 f: (818) 244-9761

Cooperating Languages
Core Language

Hesam Samimi (advised by Alan Borning)

August 23, 2012

Cooperating Languages Overview

First a quick review of *CooperatingLanguages* framework. Class definitions looked like this:

```
class Point
      Attributes
           x, y: Real
      Constraints
           x >= 0
           y >= 0
```

Fig. 1. Classes

And then specific problems could be described and solved by hosting one (and maybe more) appropriate languages.

Now Designing The Core Language

Our goal is to have a language framework with *understanding* of its own at the very bottom level. That way the programmer doesn't have to repeat (implicit) things again and again, but would be able to say the rules once and for all and be assured things work. Of course, this helps us reducing LoCs as well.

To this end we want our core language to *understand* logic. That doesn't require the ability to *search*, but simply being able to *follow the rules*. This is why we think Datalog fits the bill, because it does the above exactly, not more and not less. Therefore, while being declarative, it still scales well.

This is why we would need relations. Another advantage of relations is that they can serve as a natural bridge between the lower level imperative things and various constraint languages at the other end.

Moving onto Relations

We have the option of keeping the object oriented semantics as the core language, and then work out a translation to relations for this purpose. However, I think

that would be limiting. It wouldn't be as clean as having the relations be the foundation. For one, relations generalize classes and enable some things to be expressed more naturally such as operations on whole collection of things, e.g. composition, joins, and so on (*more to be filled in here...*).

Should we do this, class definitions become syntactic sugars for unary relations representing the class and binary relations representing the attributes. Symbol "." is a relational join operation:

```
relation Point(this: Point) →
            this.(Point_x) >= 0
            this.(Point_y) >= 0
relation Point_x(p: Point, x: Real)
relation Point_y(p: Point, y: Real)
```

Fig. 2. Relations

If `p: Point`, a field access `p.x` is a syntactic sugar for join operation `p.(Point_x)`.

Relations are just typed collections, so like any collection they can be empty or contain elements, called *tuples*. Only tuples of the right type (e.g., for `Point` relation: `(Point)`) can be added to them.

When an object is instantiated its associated class relation adds it as a new tuple.

Relations may be *extensional* or *intensional*.

Extensional relations are manually populated (e.g. tables in a database). They're denoted by "→" meaning they imply the condition specified on the right side. The condition may be just `true`.

Intensional relations are inferred (e.g. a database query). They are denoted by "←", which means they're inferred based on the condition specified on the right side.

Here is a binary intensional relation. It relates points on the same vertical line:

```
relation VLine(p: Point, q: Point) ←
            p.x = q.x
```

Fig. 3. Binary intensional relation

Here is a ternary intensional relation:

```
relation VLineH(p: Point, q: Point, h: Real) ←
            VLine(p, q)
            h = abs(p.y - q.y)
```

Fig. 4. Ternary intensional relation

Architecture

With Datalog at the core of the language, we will host an imperative language mainly used to manually instantiate and populate the extensional relations, as well as interface into the outside world.

We'll follow the FRP model, but neither the dependencies nor how to update values will be necessary to specify. User-specified relation rules already specify all that's needed.

At any given time a number of facts are known. Each fact has an implicit time step as part of the tuple. When a change occurs (new fact comes in) the Datalog engine, which is always alive at the core, incrementally produces new facts for the incremented time step. All this is implicit. (*I know this leads to more questions than insights, but we'll have to work things out gradually.*)

So now for everything expressible within Datalog rules we're good. This surprisingly covers a lot of ground, but is obviously not Turing complete. So we will occasionally be hosting some of our cooperative languages, including the imperative or the constraint solving ones (see Table 1).

Table 1. Cooperating Languages Architecture

Purpose	Level	Language	Triggering Mechanism
constraint solving	high	cooperating solvers	explicit / manual
models / relations / rules	medium-high	datalog	implicit / automatic
funtions	medium-low	functional	explicit / manual
methods	low	imperative	explicit / manual

A programming task should start at the second entry of the table above, that is defining models, relations, and rules governing them. Then as necessary hooks should be provided to call in pieces operating at higher or lower levels (see Figure 5).

Implementation

Probably not difficult. Finding or writing a relation library shouldn't be too difficult. We do need a native Datalog engine though to avoid having to do system

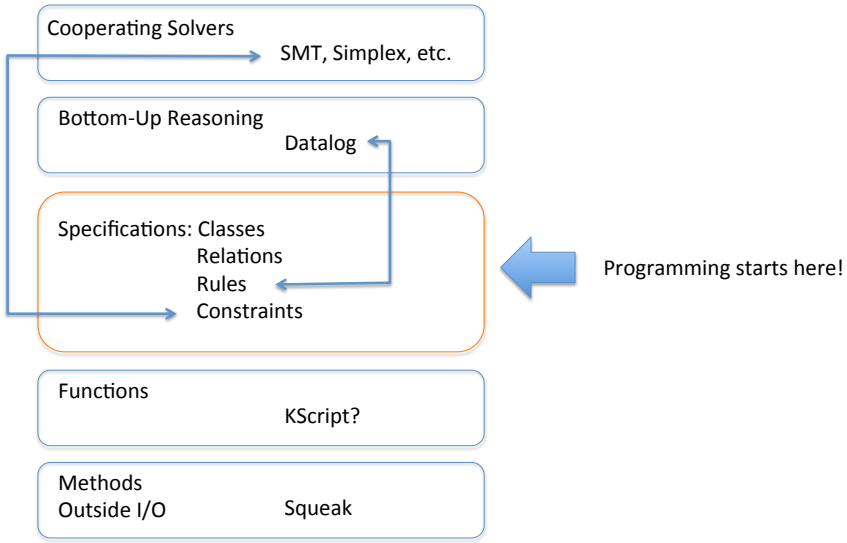

Fig. 5. Different Components of A Cooperating Languages Program

calls. We can either take a stab at implementing a naive version ourselves, or find a way to host an existing implementation. I do have the source code for a Java implementation of Datalog called Overlog (from Berkeley BOOM project).

Example

Here is an example that uses dynamic programming to compute shortest distances between all pairs of vertices in a graph [1]:

```
relation Edge(origin: Int, dest: Int, distance: Int) →
    true
relation ShortestDistance(origin: Int, dest: Int, hops: Int, distance: Int) ←
  (hops = 0 && Edge(origin, dest, distance)) ||
      (exists u: Int, v: Int, w: Int |
          (ShortestDistance(origin, hops, hops - 1, u) &&
           ShortestDistance(hops, dest, hops - 1, v) &&
           ShortestDistance(origin, dest, hops - 1, w) &&
           distance = min(u + v, w)))
```

Fig. 6. Shortest Path

This is scalable, incremental, and parallelizable, all for free, due to the Datalog semantics.

Example in Tiles

And here is how it might look like in the tile syntax that Alan K suggested:

```
relation Edge: there is an Edge between vertices origin: Int and dest: Int at a distance of distance: Int →
    true
relation ShortestDistance: ShortestDistance between vertices origin: Int and dest: Int
   with at most hops: Int hops is distance: Int ←
   (hops = 0 && there is an Edge between vertices origin and dest at a distance of distance) ||
   (exists u: Int, v: Int, w: Int |
          (ShortestDistance between vertices origin and hops with at most hops
- 1 hops is u &&
           ShortestDistance between vertices hops and dest with at most hops
- 1 hops is v &&
           ShortestDistance between vertices origin and dest with at most hops
- 1 hops is w &&
           distance = min(u + v, w)))
```

Fig. 7. Shortest Path in Tiles Datalog

References

1. U. V. Vazirani. Dynamic programming. chapter 6. http://www.cs.berkeley.edu/~vazirani/algorithms/chap6.pdf.