# Architectures for Cooperating Languages

Hesam Samimi

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-2012-001

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

# Architectures for Cooperating Languages

Hesam Samimi

hesam@cs.ucla.edu

March 28, 2012

## An Initial Proposal

Before we move onto cooperative solving, I propose we initially design an extensible multi-language paradigm, expressed in Alan B's* idea on dataflow regions to connect different parts of a program. Here one program may be composed of several sub-programs given in multiple languages, e.g. an imperative part, a linear constraint solving part, and a 1st order constraint solving part, etc. (I think even that would be impressive for demo purposes.)

– I use the term 'language' instead of 'solver', but a language can just be an API to a solver, e.g. SAT/SMT, simplex.
– Events may trigger a region (and therefore everything downstream) to re-execute.
– I propose we start by a framework that already embeds several languages, but supports new language additions.
– What the base languages have in common is that they are all founded on top of a set of primitives and relations. (Because the model of a program isn't tied to a specific language, and relations are good at modeling.)
– At the model level, relations (e.g. optional class hierarchies) can be given, and invariants can be specified (rules on tuples in a relation, or invariants for a class) using the usual relational operations.
– The base languages should include an imperative one (referred as Imp).
– I propose we also add linear solving, SMT and relational 1st order constraint solving, and Datalog.
– Values are tuples, as member of a particular relation.
– Each relation is defined by a type tuple, e.g. `link: Relation[Node,Node]`.
– Each class is just a unary relation. e.g. `class Node: Relation[Node]`.
– Each field is just a binary relation. e.g. `value: Relation[Node,Int]`.
– Each variable is a relation. e.g. `myNode: Relation[Node]`, or `myPoint: Relation[Int,Int]`. This way a variable can always store a collection of tuples, including just one.
– Base languages share primitives and relations (thus variables).
– We then allow any new language to be added, as long as an abstraction / concretization function is provided to translate state between our base relational language and this new language. E.g. javascript, prolog, etc.
– For each region it can be specified which relations are writable. Everything else is read-only for that region.

*Alan Borning, University of Washington, consulting to VPRI, Spring 2012.

– Each tuple can have two implicit parts for specifying space and time. This helps reasoning about distributed systems and temporal properties. Most programs won't care about space and time parts of tuples and just read the stored values. For example a tuple just storing a real number 4 may look like `(#Node0, 23, 4)`, which gives value 4 at a certain node in a network `#Node0` and pseudo-time 23. Programs may use space and time information directly as part of computation.
– For now we let each region explicitly specify its language.
– I propose Yoshiki to do the relational foundation, the dataflow lang, and the imperative lang, Alan B to do linear solving lang, and I do the SMT and 1st order relational constraint solving lang and Datalog.

## Examples

### Example 1: Integer Square Root

– *Model:*

```
var x: Relation[Int]
var y: Relation[Int]
```

– *Program:*

```
region1 ---> region2 ---> region3
```

– *Regions:*
*region1:*

```
language: Imp
modifies: x
program: { x = readInput }
```

*region2:*

```
language: SMT
modifies: y
program: { y >= 0 && y*y >= x && (y+1)*(y+1) < x }
```

*region3:*

```
language: Imp
program: { print y }
```

### Example 2: Graph Reachability

– *Model:*

```
class Node: Relation[Node]
var edge: Relation[Node, Node]
var reachable: Relation[Node, Node]
```

- *Program:*

```
region1 ---> region2 ---> region3
```

- *Regions:*
  *region1:*

```
    language: Imp
    modifies: Node, edge
    program: { var a = Node new
               var b = Node new
               var c = Node new
               edge add List((a,b), (b,c)) }
```

  *region2:*

```
    language: Datalog
    modifies: reachable
    program: { reachable(p, q) := edge(p, q).
               reachable(p, q) := reachable(p, r), edge(r, q). }
```

  *region3:*

```
    language: Imp
    program: { print reachable }
```

**Example 3: Hardwired Sample Circuit**

- *Model:*

```
var supply: Relation[Real]
var meter: Relation[Real]
var current: Relation[Real]
var r1: Relation[Real]
var r2: Relation[Real]
```

- *Program:*

```
region1 ---\
            ---> region3 ---> region4
region2 ---/
```

- *Regions:*
  *region1:*

```
    language: Imp
    modifies: r1, supply
    program: { r1 = readInput
               supply = readInput }
```

  *region2:*

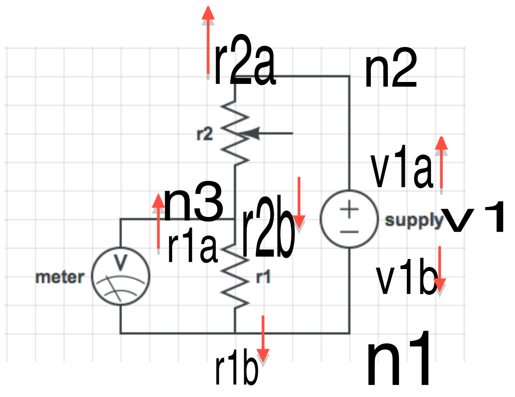

**Fig. 1.** Example 3

```
language: Imp
modifies: r2
program: { r2 = readInput }
```

*region3:*

```
language: Linear
modifies: current, meter
program: { supply = (r1 + r2) * current
           meter = current * r1 }
```

*region4:*

```
language: Imp
program: { redraw meter }
```

**Example 4: Bridge Circuit from Generic Kit**

– *Model:*

```
class Node: Relation[Node]
{
  var v: Relation[Node,Real]
  var devices: Relation[Node,Resistor]
  invariant: sum(devices.i) = 0
}

class Resistor(n1: Node, n2: Node, r: Real): Relation[Resistor]
{
  var n1, n2: Relation[Resistor,Node]
  var r, i: Relation[Resistor,Real]
  invariant: n1.v - n2.v = i * r
}
```

```
class Meter(n1: Node, n2: Node, r: Real): Relation[Meter]
  extends Resistor
{
  invariant: i = 0
}

class Supply(n1: Node, n2: Node, r: Real, val: Real): Relation[Meter]
  extends Resistor
{
  var val: Relation[Supply,Real]
  invariant: n1.v - v2.v = val
}

class Ground: Relation[Ground] extends Node
{
  invariant: v = 0
}
```

– *Program:*

```
region1 ---> region2 ---> region3 ---> region4
```

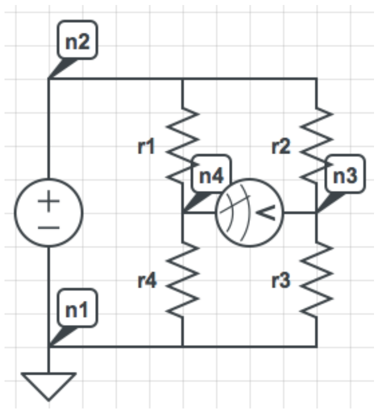

**Fig. 2.** Example 4

– *Regions:*
*region1:*

```
  language: Imp
  modifies: Ground, Node, Supply, Resistor, Meter, devices
```

```
    program: { var n1 = Ground new
               var n2 = Node new
               var n3 = Node new
               var n4 = Node new
               var supply = Supply new (n1, n2, _, _)
               var r1 = Resistor new (n4, n2, _)
               var r2 = Resistor new (n2, n3, _)
               var r3 = Resistor new (n3, n1, _)
               var r4 = Resistor new (n1, n4, _)
               var meter = Meter new (n4, n3, _)
               devices add List((n1, supply), (n1, r3), (n1, r4),
                                (n2, supply), (n2, r1), (n2, r2),
                                (n3, meter), (n3, r2), (n3, r3),
                                (n4, r1), (n4, meter), (n4, r4)) }
```

*region2: (*implicit?)*

```
    language: Imp
    program: { <collect all invariants for region3 (how?)> }
```

*region3:*

```
    language: Linear
    modifies: v, i
    program: { <solve invariants... (how?)> }
```

*region4:*

```
    language: Imp
    program: { redraw meter }
```