Experience with and Plans for Rascal

a DSL for software analysis and transformation

Mark Hills, Paul Klint, Tijs van der Storm & Jurgen Vinju
Supporting Modular, Extensible & Efficient Equation Solving in Rascal
Rascal is a JVM language suitable for

- Meta-programming
- Software analysis & transformation
- Compiler construction
- Applications: software metrics, refactoring, repository analysis, code generation
- Design & implementation Domain-Specific Languages
- Applications: gaming, questionnaires, banking, tax regulation, laws and legal analysis, …
Rascal Features

- Sophisticated built-in datatypes: list, set, relation, listrelation, map, datetime, location, …

- Immutable values, but mutable variables (references to immutable values)

- Static types with local type inference

- Pattern matching on all values

- (Higher order) functions using pattern-based dispatch

- Visiting/traversing values

- Syntax definitions and parsing

- Concrete syntax values

- Familiar (Java-like) syntax

- Compiled to JVM byte code

- Java and Eclipse integration

- Command line (REPL)
Rascal Applications

• Compiler for Numerical Simulation Language (Magnolia)
• Compiler for GPU language
• Rascal to JVM compiler
• Software metrics (OSSMETER)
• PHP analysis (PHP AiR)
• Java Refactoring
• Hibernate performance analysis
• Javascript analysis & transformation

DSLs for
• Digital Forensics
• Financial Transactions
• Game Economics
• Tax Forms
• Accountancy
• Legal rules
Rascal Ecosystem

- Rascal Language
- Rascal interpreter
- Eclipse integration
- Rascal Libraries
  - Data types, statistics, …
- Command Line Interface (REPL)

- Rascal to JVM compiler (uses coroutines internally to implement pattern matching)
- Compiler-based REPL
Today’s Topic: solving equations

- Rascal already provides everything you need for static analysis.

- From day one Rascal has supported a `solve` statement to support fixed point equations. Used in many applications with acceptable performance.

- Now we want to
  - modularize the solve statement to support large sets of equations
  - use/adapt/develop more efficient solution techniques
Example 1: Transitive Closure (datalog)

Given a binary relation $r$, its transitive closure can be defined as follows:

\[
\begin{align*}
\text{trans}(e_1, e_2) & : \ r(e_1, e_2). \\
\text{trans}(e_1, e_3) & : \ r(e_1, e_2), \ \text{trans}(e_2, e_3).
\end{align*}
\]
Example 1: Transitive Closure (Rascal `solve`)

```rascal
rel[int,int] trans(rel[int,int] r) {
    rel[int,int] t = r;
    solve (t) {
        t += (t o r);
    }
    return t;
}
```

Initialize `t` to `r`

Iterate until fixed point of `t` is reached

Composition

```rascal
while (t != t1) {
    t1 = t;
    t += (t o r);
}
```
Aside

• This is just a very simple example to illustrate
  solve

• Rascal has very good built-in support for
  transitive closure, reachability, and other
  relational algebra operators

• Datalog approach is based on implications and a
  search procedure

• Rascal approach is constructive & programmable
Example 2: Dataflow equations (Rascal `solve`)

```rascal

    set[stat] STATEMENT = carrier(PRED);
    rel[stat,def] DEF = definition(DEFS);
    rel[stat,def] USE = use(USES);

    rel[stat,def] LIN = {};
    rel[stat,def] LOUT = DEF;

    solve (LIN, LOUT) {
        LIN = { <S, D> | stat S <- STATEMENT, def D <- USE[S] + (LOUT[S] - (DEF[S])) };
        LOUT = { <S, D> | stat S <- STATEMENT, stat Succ <- successors(PRED,S),
                     def D <- LIN[Succ] };
    }
    return LIN;
}
```

Dragon book solution
Example 3: Expression Simplification in PHP AiR

Apply available normalization functions to simplify an expression:

```cpp
Expr simplifyExpr(Expr e, loc baseLoc) {
    e = normalizeConstCase(inlineMagicConstants(e, baseLoc));
    solve(e) {
        e = algebraicSimplification(simulateCalls(e));
    }
    return e;
}
```

See: Hills, Klint, & Vinju: Static, Lightweight Includes Resolution for PHP, ASE, 2014

Uses a 4.5 MLOC PHP corpus that has now been extended to 27.5 MLOC (not yet published work)
General Format of `solve`

\[
\begin{align*}
    r_1 &= \text{init}_1; \\
    \quad \vdots \\
    r_n &= \text{init}_n; \\
    \text{solve} \ (r_1, \ldots, r_n) \{ \\
    &\quad r_1 = \{<x,y> \mid <x,y> \leftarrow r_1, c_1(r_1, \ldots, r_n)\} \\
    &\quad \cdots \\
    &\quad r_n = \{<x,y> \mid <x,y> \leftarrow r_n, c_n(r_1, \ldots, r_n)\} \\
    \} \\
\end{align*}
\]

We can also specify an upper bound on the number of iterations here.
# Assessment of `solve`

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Readability</strong></td>
<td>Good, declarative, high abstraction level</td>
</tr>
<tr>
<td><strong>Direction</strong></td>
<td>Solutions can grow or shrink towards a solution</td>
</tr>
<tr>
<td><strong>Information Use</strong></td>
<td>Completely open, any visible variable (bound to AST, table, auxiliary relation, …) may be queried</td>
</tr>
<tr>
<td><strong>Complexity</strong></td>
<td>Turing complete, termination not guaranteed (but # of iterations can be restricted), speed in EXP</td>
</tr>
<tr>
<td><strong>Safety</strong></td>
<td>Immutable data</td>
</tr>
<tr>
<td><strong>Algorithm</strong></td>
<td>Brute force, visit all elements in each iteration</td>
</tr>
<tr>
<td><strong>Modularity</strong></td>
<td>Bad, <code>solve</code> imposes a lexical scope for mutually recursive relations</td>
</tr>
</tbody>
</table>
How to achieve Modular, Extensible, Efficient Equation Solving in Rascal?
(A sample of) Related Work


- Whaley, Avots, Carbin & Lam, Using Datalog with Binary Decision Diagrams for *Program Analysis*, 2005


- O. de Moor et al., *QL* Object-oriented Queries made Easy, 2007

- M. Bravenboer and Y Smaragdakis., Exception analysis and points-to analysis: better together. 2009


Design Considerations

• Profit from the success of Datalog variants in static analysis

• Integrate with Rascal’s immutable values, mostly functional semantics and syntactic style

• Support open extension and modularity for solve

• Create a good match with efficient implementation techniques (e.g., finite differencing, magic sets, BDDs, TrieJoin, …)

• Disclaimer: first, exploratory, ideas!
Steps Towards an open, modular **solve** statement

```rascal
rel[int,int] trans(rel[int,int] r) {
    rel[int,int] t = r;
    solve (t) {
        t += (t o r);
    }
    return t;
}
```

Given the context `r` solve `t`:

- `fix rel[int,int] t(rel[int,int] r);`
- `fix rel[int,int] t() = r;`
- `fix rel[int,int] t() += t() o r;`

Declare context values. Together with the variable name this **identifies** a specific fixed point computation.

Initial value of `t`

Increments to `t`
A classic PointsTo Analysis in a DataLog (bddbddd)

2.2 Example

Algorithm 1  Context-insensitive points-to analysis with a precomputed call graph, where parameter passing is modeled with assignment statements.

Domains

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>V</td>
<td>262144</td>
<td>variable.map</td>
</tr>
<tr>
<td>H</td>
<td>65536</td>
<td>heap.map</td>
</tr>
<tr>
<td>F</td>
<td>16384</td>
<td>field.map</td>
</tr>
</tbody>
</table>

Relations

- input \( vP_0 \) (variable : V, heap : H)
- input \( store \) (base : V, field : F, source : V)
- input \( load \) (base : V, field : F, dest : V)
- input \( assign \) (dest : V, source : V)
- output \( vP \) (variable : V, heap : H)
- output \( hP \) (base : H, field : F, target : H)

Rules

1. \( vP(v, h) \) : \( vP_0(v, h) \).
2. \( vP(v_1, h) \) : \( assign(v_1, v_2), vP(v_2, h) \).
3. \( hP(h_1, f, h_2) \) : \( store(v_1, f, v_2), vP(v_1, h_1), vP(v_2, h_2) \).
4. \( vP(v_2, h_2) \) : \( load(v_1, f, v_2), vP(v_1, h_1), hP(h_1, f, h_2) \).

See: Whaley, Avots, Carbin & Lam, Using Datalog with Binary Decision Diagrams for Program Analysis, 2005
PointsTo: classic solve

alias VP = rel[V variable, H heap];
alias HP = rel[H base, F field, H target];
alias FS = rel[V base, F field, H target];
alias FL = rel[V base, F field, V destination];
alias ASG = rel[V dest, V source];

tuple[VP vP, HP hP] pointsTo1(VP vP0, FS store, FL load, ASG assign){

  vP = vP0;
  hP = {};

  solve(vP, hP){

    vP += { <v1, h> | <v1, v2> <- assign, <v2, h> <- vP };

    hP += { <h1, f, h2> | <v1, f, v2> <- store, <v1, h1> <- vP, <v2, h2> <- vP };

    vP += { <v2, h2> | <v1, f, v2> <- load, <v1, h1> <- vP, <h1, f, h2> <- hP };
  }

  return <vP, hP>;

}

(1) vP(v, h) :- vP0(v, h).
(2) vP(v1, h) :- assign(v1, v2), vP(v2, h).
(3) hP(h1, f, h2) :- store(v1, f, v2), vP(v1, h1), vP(v2, h2).
(4) vP(v2, h2) :- load(v1, f, v2), vP(v1, h1), hP(h1, f, h2).

Convenience abbreviations for some types
PointsTo: open & modular

```rascal
fix VP vP(VP vP0, FS store, FL load, ASG assign);
fix HP hP(VP vP0, FS store, FL load, ASG assign);

fix VP vP() = vP0; // (1)
fix HP hP() = {};

fix VP vP() += { <v1, h> | <v1, v2> <- assign, <v2, h> <- vP()}; // (2)
fix HP hP() += { <h1, f, h2> | <v1, f, v2> <- store, 
                  <v1, h1> <- vP(), <v2, h2> <- vP() }; // (3)

fix VP vP() += { <v2, h2> | <v1, f, v2> <- load, 
                  <v1, h1> <- vP(), <h1, f, h2> <- hP() }; // (4)

tuple[VP vP, HP hP] pointsTo2(VP vP0, FS store, FL load, ASG assign) = 
  <vp(vP0, store, load, assign), hp(vP0, store, load, assign)>;
```
Discussion

• Initial experiment, expect further syntactic/semantic improvements

• We are extending our capsule immutable collections library* with bidirectional multi-maps to support incremental computation (in finite differencing style) on binary relations

• Still open: what implementation technique is best suited?

(*) See Steindorfer & Vinju, Optimizing Hash-array Mapped Tries for Fast and Lean Immutable JVM Collections, 2015
You are invited to join!

If you want to learn more about Rascal:
●  http://www.rascal- mpl.org
●  http://tutor.rascal- mpl.org
●  http://stackoverflow.com/questions/tagged/rascal

If you are interested in source code:
https://github.com/usethesource/rascal

If you want to give us feedback:
●  hillsma@ecu.edu
●  {Paul.Klint,T.van.der.Storm,Jurgen.Vinju}@cwi.nl