#### **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 A One-Stop Shop for ... 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

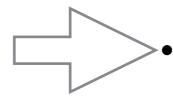
DSLs for

- Digital Forensics
- Financial Transactions
- Game Economics
- Tax Forms
- Accountancy
- Legal rules

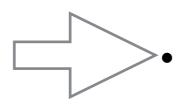


#### Rascal Ecosystem

- 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:

```
trans(e1, e2) :- r(e1, e2).
trans(e1, e3) :- r(e1, e2), trans(e2, e3).
```



## Example 1: Transitive Closure (Rascal solve)

```
Initialize t to r
rel[int,int] trans(rel[int,int] r){
                                              Iterate until fixed point
  rel[int,int] t = r;
                                              of t is reached
  solve (t) {
    t += (t o r);
                                    t1 = {};
              Composition
                                    while (t != t1) {
  return t;
                                         t1 = t;
                                         t += (t \circ 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)

```
rel[stat,def] liveVariables(rel[stat,var] DEFS, rel[stat, var] USES, rel[stat,stat] PRED){
   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),</pre>
                            def D <- LIN[Succ] };</pre>
                                                     for each block B do in[B] := \emptyset;
                                                        while changes to any of the in's occur do
   return LIN;
                                                            for each block B do begin
                                                                in[B] := use[B] \cup (out[B] - def[B])
                              Dragon book solution
```



Fig. 10.33. Live variable calculation.

## Example 3: Expression Simplification in PHP AiR

Apply available normalization functions to simplify an expression:

```
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

```
We can also specify an
                                              upper bound on the number
r_1 = init_1;
                                                     of iterations here
r_n = init_n;
solve (r_1, ..., r_n) {
   r_1 = \{ \langle x, y \rangle \mid \langle x, y \rangle \langle -r_1, c_1(r_1, \ldots, r_n) \}
   r_n = \{ \langle x, y \rangle \mid \langle x, y \rangle \langle -r_n, c_n(r_1, \dots, r_n) \}
```



#### Assessment of solve

Readability

Good, declarative, high abstraction level

Direction

Solutions can grow or shrink towards a solution

Information Use

Completely open, any visible variable (bound to AST, table, auxiliary relation, ...) may be queried

Complexity

Turing complete, termination not guaranteed (but # of iterations can be restricted), speed in EXP

Safety

Immutable data

**Algorithm** 

Brute force, visit all elements in each iteration

**Modularity** 

Bad, solve imposes a lexical scope for mutually recursive relations



### How to achieve Modular, Extensible, Efficient Equation Solving in Rascal?



#### (A sample of) Related Work

- Paige & Koenig, Finite Differencing of Computable Expressions, 1982.
- Whaley, Avots, Carbin & Lam, Using Datalog with Binary Decision Diagrams for *Program Analysis*, 2005
- Liu & Stoller, Dynamic Programming via Static Incrementalization, 2008.
- 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
- T. Veldhuizen, Leapfrog triejoin: A simple worst-case optimal join algorithm, 2012
- M. Arntzenius & N. R.Krishnaswami, Datafun: a Functional Datalog, 2016



#### 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

```
rel[int,int] trans(rel[int,int] r){
    rel[int,int] t = r;

    solve (t) {
        t += (t o r);
    }

        return t;
}
Declare context values.
Together with the variable name this identifies a specific fixed point computation

fix rel[int,int] t(rel[int,int] r);

fix rel[int,int] t() = r;
    Initial value of this identifies a specific fixed point computation

Increments to the identifies a specific fixed point computation

Together with the variable name this identifies a specific fixed point computation

Increments to the identifies a specific fixed point computation

Together with the variable name this identifies a specific fixed point computation

Increments to the identifies a specific fixed point computation

Together with the variable name this identifies a specific fixed point computation

Increments to the identifies a specific fixed point computation

Increments to the identifies a specific fixed point computation

Together with the variable name this identifies a specific fixed point computation

Together with the variable name this identifies a specific fixed point computation

Increments to the identifies a specific fixed point computation

Together with the variable name this identifies a specific fixed point computation

Together with the variable name this identifies a specific fixed point computation

Increments to the identifies a specific fixed point computation

Increments to the identifies a specific fixed point computation

Increments to the identifies a specific fixed point computation

Together with the variable name this identifies a specific fixed point computation

Together with the variable name this identifies a specific fixed point computation

Increments to the identifies a specific fixed point computation

Together with the variable name this identifies a specific fixed point computation

Together with the variable name this identifies a specific fixed point computation

Increments the identifies a specific fixed point computation

Together with the variable na
```

Given the context r solve t



rel[int,int] trans(rel[int,int] r) = t(r);

#### A classic Points To Analysis in a DataLog (bddbddb)

#### **Example** 2.2

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

```
DOMAINS
                                           See: Whaley, Avots, Carbin & Lam,
                                           Using Datalog with Binary Decision Diagrams for
      V
              262144 variable.map
      \mathbf{H}
                                           Program Analysis, 2005
              65536
                      heap.map
      \mathbf{F}
                      field.map
              16384
```

#### RELATIONS

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

#### 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).
   vP(v_2,h_2) : -load(v_1,f,v_2), vP(v_1,h_1), hP(h_1,f,h_2).
```



#### PointsTo: classic solve

```
alias VP = rel[V variable, H heap];
alias HP = rel[H base, F field, H target];
                                                                      Convenience abbreviations for
alias FS = rel[V base, F field, V source];
                                                                               some types
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){
                    (1) vP(v, h) :- vP0(v, h).
    vP = vP0;

hP = \{\};
                                                (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).
    solve(vP, hP){
        vP += \{ < v1, h >
                                 | <v1, v2> <- assign, <v2, h> <- vP };</pre>
        hP += \{ \langle h1, f, h2 \rangle \mid \langle v1, f, v2 \rangle \langle -store, \langle v1, h1 \rangle \langle -vP, \langle v2, h2 \rangle \langle -vP \};
                                 | <v1, f, v2> <- load, <v1, h1> <- vP, <h1, f, h2> <- hP };</pre>
        vP += \{ < v2, h2 >
    return <vP, hP>;
                                                 (4) vP(v2, h2) :- load(v1, f, v2), vP(v1, h1), hP(h1, f, h2).
```

#### PointsTo: open & modular

```
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;

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)
```



#### 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: <a href="https://github.com/usethesource/rascal">https://github.com/usethesource/rascal</a>

If you want to give us feedback:

- hillsma@ecu.edu
- {Paul.Klint, T. van.der.Storm, Jurgen. Vinju}@cwi.nl

