Rascal-MPL
A Short Introduction

Rodrigo Bonifácio

September 23, 2019

Rascal is an experimental domain specific language for metaprogramming, such as static code analysis, program transformation and implementation of domain specific languages. It is a general meta language in the sense that it does not have a bias for any particular software language.

Wikipedia

Domain

Rascal-MPL allows a developer to create full-fledged program analysis and manipulation tools, including support for syntax definition and strategic programming.

Features

- hybrid programming model (imperative and functional styles)
- immutable data structures and support for generic types
- pattern matching and visitors using different strategies
- syntax definitions and parsing + REPL support
- Java and Eclipse integration

Section 01:

Section 01:

- Exploring Rascal Concepts using Colored Trees

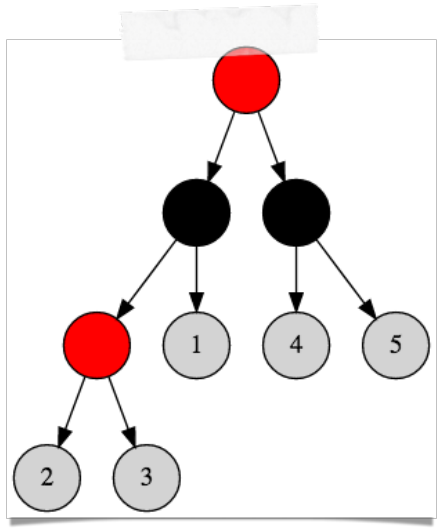
Section 01:

- Exploring Rascal Concepts using Colored Trees

The following slides are based on the P. Klint's Introductory Course on Rascal

```
1  module ColoredTrees
2
3  data ColoredTree = Leaf(int N)
4                   | Red(CTree left, CTree right)
5                   | Black(CTree left, CTree right)
6                   ;
7
8
9  ColoredTree sample = Red(Black(Leaf(1), Red(leaf(2), Leaf(3))),
10                        Black(Leaf(4), Leaf(5)));
```

```
1  module ColoredTrees
2
3  data ColoredTree = Leaf(int N)
4                   | Red(CTree left, CTree right)
5                   | Black(CTree left, CTree right)
6                   ;
7
8
9  ColoredTree sample = Red(Black(Leaf(1), Red(leaf(2), Leaf(3))),
10                          Black(Leaf(4), Leaf(5)));
```



Pattern Matching (switch-case)

```
1  // number of nodes
2  int elements(ColoredTree tree) {
3    switch(tree) {
4      case Leaf(n) : return 1;
5      case Black(l, r): return 1 + elements(l) + elements(r);
6      case Red(l, r) : return 1 + elements(l) + elements(r);
7    }
8    return res;
9  }
```

Pattern Matching (visitor to collect information)

```
1  // total of values in a tree (imperative style)
2  int sumTree(ColoredTree tree) {
3    int res = 0;
4    top-down visit (tree) {
5      case Leaf(n): res = res + n;
6    }
7    return res;
8  }
```

- Traversal strategies
  - top-down visit (top-down-break)
  - bottom-up visit (bottom-up-break)
  - innermost
  - outermost

Pattern Matching (visitor to transform a tree)

```
1    // increment all values in a tree
2    CTree increment(ColoredTree tree) =
3      top-down visit (tree) {
4        case Leaf(n) => leaf(n + 1)
5      };
```

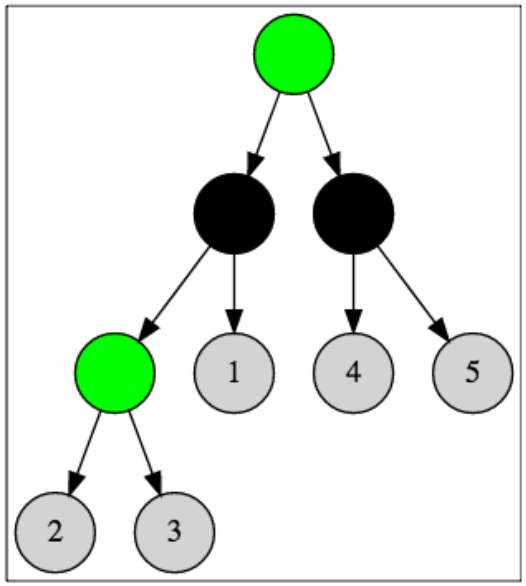
Pattern Matching (visitor to transform a tree)

```
1  // increment all values in a tree
2  CTree increment(ColoredTree tree) =
3    top-down visit (tree) {
4      case Leaf(n) => leaf(n + 1)
5    };
```

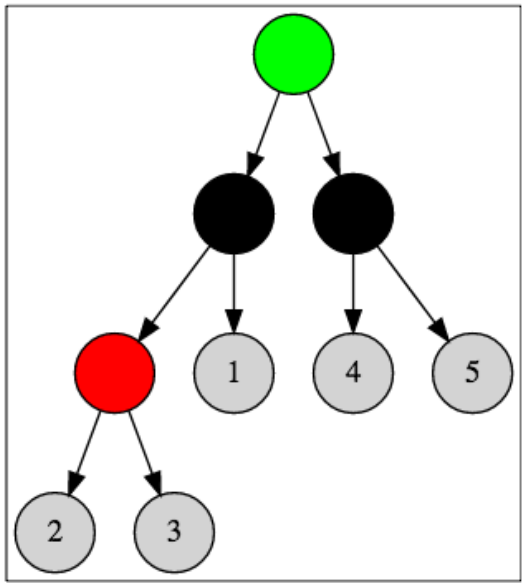
Change all reds to green

```
1  data ColoredTree = Green(ColoredTree l, ColoredTree r);
2  ColoredTree allRedsToGreen(ColoredTree tree) =
3    top-down-break visit (tree) {
4      case Red(l, r) => Green(l, r)
5    };
```

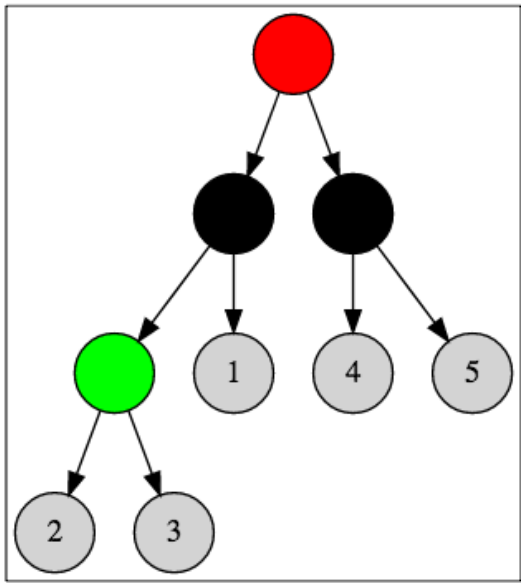
top-down



top-down-break



bottom-up-break



Comprehension

```
GHCi, version 7.10.3: http://www.haskell.org/ghc/  :? for help
Prelude> let list = [1..10]
Prelude> [x * x | x <- list, even x]
[4,16,36,64,100]
```

Comprehension

```
GHCi, version 7.10.3: http://www.haskell.org/ghc/  :? for help
Prelude> let list = [1..10]
Prelude> [x * x | x <- list, even x]
[4,16,36,64,100]
```

Comprehension is available in many programming languages

Comprehension

```
GHCi, version 7.10.3: http://www.haskell.org/ghc/  :? for help
Prelude> let list = [1..10]
Prelude> [x * x | x <- list, even x]
[4,16,36,64,100]
```

Comprehension is available in many programming languages, including Haskell, Python, Rascal, and so on.

Comprehension

```
GHCi, version 7.10.3: http://www.haskell.org/ghc/  :? for help
Prelude> let list = [1..10]
Prelude> [x * x | x <- list, even x]
[4,16,36,64,100]
```

Comprehension is available in many programming languages, including Haskell, Python, Rascal, and so on.

Rascal goes further, combining comprehension and generic traversals

```
1  rascal> [n | /leaf(int n) <- sample()];
2  list[int]: [1,2,3,4,5]
```

Exercise 01

- implement a function that converts all `red` nodes into `green` nodes
- implement a function that converts only the top-level `red` nodes into `green` nodes
- reimplement `elements` and `sumTree` using a more functional style

String Interpolation: Export to DOT Notation (1/3)

```
1    str export(CTree t) {
2      str g = "digraph g { \n";
3        int id = 0;
4        map[CTree, str] decls = ();
```

String Interpolation: Export to DOT Notation (2/3)

```
1        // print the nodes
2        top-down visit(t) {
3          case red(l, r) :
4          {
5            id = id + 1;
6            g += "  n<id> [label = \"\" shape = circle fillcolor = red style = filled]\n";
7            decls += (red(l,r) : "n<id>");
8          }
9          case black(l, r) :
10         {
11           id = id + 1;
12           g += "  n<id> [label = \"\" shape = circle fillcolor = black style = filled]\n";
13           decls += (black(l,r) : "n<id>");
14         }
15         case leaf(n) :
16         {
17           id = id + 1;
18           g = g + "  n<id> [label = <n>] \n";
19           decls += (leaf(n) : "n<id>");
20         }
21       };
```

String Interpolation: Export to DOT Notation (3/3)

```
1         // print the edges
2         top-down visit(t) {
3           case red(l, r) : {
4             g += "  <decls[red(l,r)]> -\> <decls[l]> \n";
5             g += "  <decls[red(l,r)]> -\> <decls[r]> \n";
6           }
7           case black(l, r) : {
8             g += "  <decls[black(l,r)]> -\> <decls[l]> \n";
9             g += "  <decls[black(l,r)]> -\> <decls[r]> \n";
10          }
11        }
12
13        g += "}";
14
15      return g;
16    }
```

Section 02:

- Hands on: Scrap Your Boilerplate

Scrap Your Boilerplate: A Practical Design Pattern for Generic Programming

Ralf Lämmel
Vrije Universiteit, Amsterdam

Simon Peyton Jones
Microsoft Research, Cambridge

Abstract

We describe a design pattern for writing programs that traverse data structures built from rich mutually-recursive data types. Such programs often have a great deal of "boilerplate" code that simply walks the structure, hiding a small amount of "real" code that constitutes the reason for the traversal.

Our technique allows most of this boilerplate to be written once and for all, or even generated mechanically, leaving the programmer free to concentrate on the important part of the algorithm. These generic programs are much more adaptive when faced with data structure evolution because they contain many fewer lines of type-specific code.

Our approach is simple to understand, reasonably efficient, and it handles all the data types found in conventional functional programming languages. It makes essential use of rank-2 polymorphism, an extension found in some implementations of Haskell. Further it relies on a simple type-safe cast operator.

Categories and Subject Descriptors

D.3.1 [**Programming Languages**]: Formal Definitions and Theory; D.2.13 [**Software Engineering**]: Reusable Software

General Terms

Design, Languages

Keywords

Generic programming, traversal, rank-2 types, type cast

1 Introduction

Suppose you have to write a function that traverses a rich, recursive data structure representing a company's organisational structure, and increases the salary of every person in the structure by 10%. The interesting bit of this algorithm is performing the salary-increase — but the code for the function is probably dominated by "boilerplate" code that recurses over the data structure to find the specified department as spelled out in Section 2. This is not an unusual situation. On the contrary, performing queries or transformations over rich data structures, nowadays often arising from XML schemata, is becoming increasingly important.

Boilerplate code is tiresome to write, and easy to get wrong. Moreover, it is vulnerable to change. If the schema describing the company's organisation changes, then so does every algorithm that recurses over that structure. In small programs which walk over one or two data types, each with half a dozen constructors, this is not much of a problem. In large programs, with dozens of mutually recursive data types, some with dozens of constructors, the maintenance burden can become heavy.

Generic programming techniques aim to eliminate boilerplate code. There is a large literature, as we discuss in Section 9, but much of it is rather theoretical, requires significant language extensions, or addresses only "purely-generic" algorithms. In this paper, we present a simple but powerful design pattern for writing generic algorithms in the strongly-typed lazy functional language Haskell. Our technique has the following properties:

- It makes the application program adaptive in the face of data type (or schema) evolution. As the data types change, only two functions have to be modified, and those functions can easily be generated because they are not application-specific.
- It is simple *and* general. It copes with arbitrary data-type structure without fuss, including parameterised, mutually-recursive, and nested types. It also subsumes other styles of generic programming such as term rewriting strategies.
- It requires two extensions to the Haskell type system, namely (a) rank-2 types and (b) a form of type-coercion operator. However these extensions are relatively modest, and are independently useful; they have both been available in two popular implementations of Haskell, GHC and Hugs, for some time.

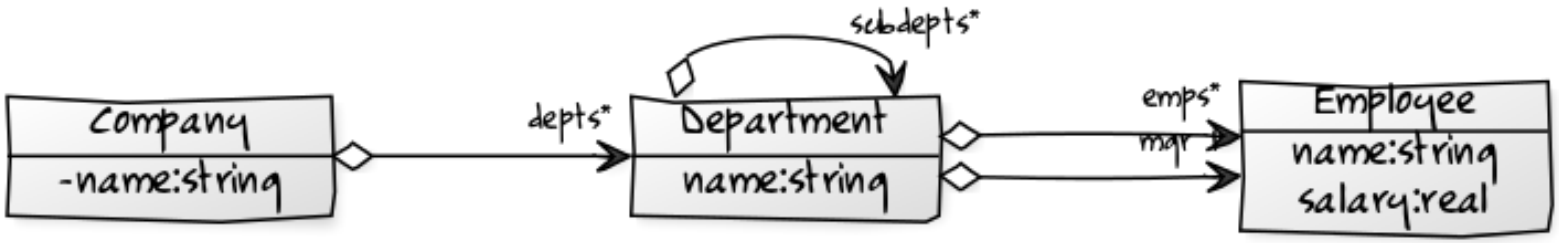
Our contribution is one of synthesis: we put together some relatively well-understood ideas (type-safe cast, one-layer maps) in an innovative way, to solve a practical problem of increasing importance. The paper should be of direct interest to programmers, and library designers, but also to language designers because of the further evidence for the usefulness of rank-2 polymorphic types.

The code for all the examples is available online at:

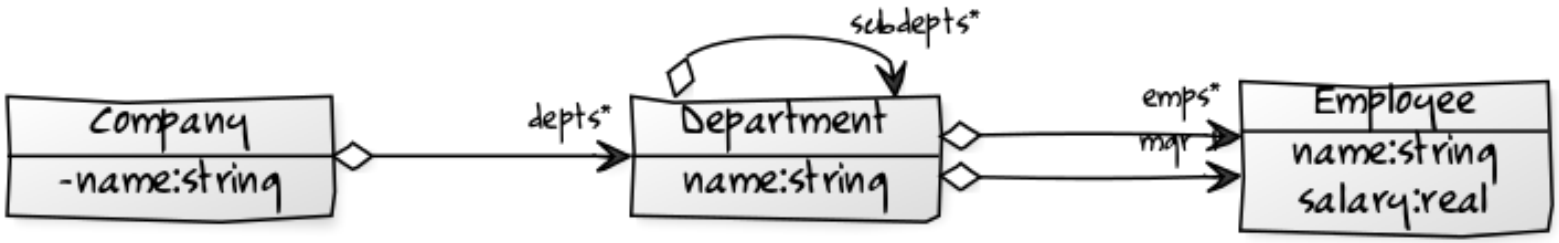
`http://www.cs.vu.nl/Strafunski/gmap/`

The distribution comes with generative tool support to generate all datatype-specific boilerplate code. Our benchmarks show that it is possible to get the run-time performance of typical generic programs reasonably close to the hand-coded boilerplate-intensive counterparts (Section 10).

Problem: generic traversals



Problem: generic traversals



Development task . . .

- compute the total of salaries in a company
- cut the salaries of a company by a given percentage

Domain model in Haskell

```haskell
module Companies where

type Name    = String
type Salary  = Double
type Manager = Employee

data Company    = C Name [Department]
data Department = D Name Manager [Department] [Employee]
data Employee   = E Name Salary
```

Cutting salaries in Haskell

```haskell
cut :: Double -> Company -> Company
cut k (C n depts) = C n (map (cutD k) depts)

cutD :: Double -> Department -> Department
cutD k (D n mgr subdepts emps) = D n (cutE k mgr) (map (cutD k) subdepts) (map (cutE k) emps)

cutE :: Double -> Employee -> Employee
cutE k (E n s) = E n (s - (k * s))
```

Total salaries in Haskell

```haskell
total :: Company -> Salary
total (C _ depts) = sum (map totalD depts)

totalD :: Department -> Salary
totalD (D _ mgr subdepts emps) = salary mgr
                               + sum (map totalD subdepts)
                               + sum (map salary emps)

salary :: Employee -> Salary
salary (E _ s) = s
```

Let's implement this domain using Rascal-MPL

Domain Model

```
1  alias Name    = str;
2  alias Salary  = real;
3  alias Manager = Employee;
4
5  data Company = company(Name n, list[Department] deps);
6  data Department = department(Name name, Manager mgr, list[Department] subdeps, list[Employee] emps);
7  data Employee = employee(Name name, Salary salary);
```

Test Cases

```
1  Employee ralf = employee("ralf", 8000.0);
2  Employee joost = employee("joost", 1000.0);
3  Employee marlow = employee("marlow", 2000.0);
4  Employee blair = employee("blair", 100000.0);
5
6  Department research = department("research", ralf, [], [joost, marlow]);
7  Department strategy = department("strategy", blair, [], []);
8
9  public Company genCom = company("genCom", [research, strategy]);
10
11 test bool testTotal() = 111000.0 == total(genCom);
12
13 test bool testIncrease() = 122100.0 == increase(0.1, genCom);
```

Exercise 02

Implement the following functions:

- increase the salary of all managers
- find the highest salary of a company
- find the salary of a given employee

Section 03:

- Exploring Syntax Definition, Parsing, and Pattern Matching on Concrete Syntax

Syntax Definition

- converts a grammar into a SGLR parser
- main components (context free grammar)
  - syntax expressed using production rules
  - lexical, layout, and keyword definitions

Syntax Definition

- converts a grammar into a SGLR parser
- main components (context free grammar)
  - syntax expressed using production rules
  - lexical, layout, and keyword definitions
- challenge: disambiguation

Let's consider a language for FSMs

```
1  module lang::fsm::AbstractSyntax
2
3  data StateMachine
4    = fsm(list[State] states, list[Transition] transitions);
5
6  data State
7    = state(str name)
8    | startState(str name)
9    ;
10
11 data Transition
12   = transition(str source, Event event, str target);
13
14 data Event
15   = event(str evt)
16   | eventWithAction(str evt, str action)
17   ;
```

Example of a FSM specification

```
1  initial state locked {
2    ticket/collect -> unlocked;
3    pass/alarm -> exception;
4  }
5
6  state unlocked {
7    ticket/eject;
8    pass -> locked;
9  }
10
11 state exception {
12   ticket/eject;
13   pass;
14   mute;
15   release -> locked;
16 }
```

A grammar definition for FSMs using Rascal

```
1  module lang::fsm::ConcreteSyntax
2
3  start syntax FSM = CState* states;
4
5  syntax CState
6    = initialState: "initial" "state" Id "{" CEvent* "}"
7    | basicState: "state" Id "{" CEvent* "}"
8    ;
9
10 syntax CEvent
11   = basicEvent: Id ("-\>" Id)? ";"
12   | actionEvent: Id "/" Id ("-\>" Id)? ";"
13   ;
14
15 lexical Id = [a-zA-Z][_a-zA-Z0-9]*;
16
17 lexical Comment = "//" ![\n]* [\n];
18
19 lexical Spaces = [\n\r\f\t\ ]*;
20
21 layout Layout = Spaces
22               | Comment
23               ;
24
25 keyword Keywords = "state" | "initial" ;
```

A parser from Concrete Syntax to Abstract Syntax (1/2)

```
1  module lang::fsm::Parser
2
3  import ParseTree;
4  import IO;
5
6  import lang::fsm::ConcreteSyntax;
7  import lang::fsm::AbstractSyntax;
8
9  public StateMachine parseFSM(str f) {
10  loc file = |file:///| + f;
11  list[State] states = [];
12  list[Transition] transitions = [];
13
14  start[FSM] parseResult = parse(#start[FSM], file);
15
16  top-down visit (parseResult) {
17     case (CState)`initial state <Id id> { <CEvent* evts>}`: {
18       states += startState(unparse(id));
19       transitions += parseEvents(id, evts);
20     }
21     case (CState)`state <Id id> { <CEvent* evts>}`: {
22       states += state(unparse(id));
23       transitions += parseEvents(id, evts);
24     }
25  };
26
27  return fsm(states, transitions);
28 }
```

A parser from Concrete Syntax to Abstract Syntax (2/2)

```
1  list[Transition] parseEvents(Id source, CEvent* evts) {
2    list[Transition] ts = [];
3    top-down visit(evts) {
4      case (CEvent)‘<Id e> -\> <Id target>;‘ :
5        ts += transition(unparse(source), event(unparse(e)), unparse(target));
6      case (CEvent)‘<Id e>;‘ :
7        ts += transition(unparse(source), event(unparse(e)), unparse(source));
8      case (CEvent)‘<Id e> / <Id a> -\> <Id target>;‘ :
9        ts += transition(unparse(source), eventWithAction(unparse(e), unparse(a)), unparse(target));
10     case (CEvent)‘<Id e> / <Id a>;‘ :
11       ts += transition(unparse(source), eventWithAction(unparse(e), unparse(a)), unparse(source));
12   }
13   return ts;
14 }
```

Exercise 03

Implement the following functions on FSMs

- interpretation, code generation, and visualization
- wfr: single start state, deterministic transitions, reachable states

Program Analysis

Counting atoms of confusion (using concrete syntax)

```
1  void analyse(CompilationUnit unit) {
2    top-down visit(unit) {
3      case (Expression)‘<OctaNumeral n>‘:
4        recordAtom("LiteralEncoding");
5
6      case (Expression)‘<Identifier id>++‘:
7        recordAtom("PostIncrement");
8
9      case (Expression)‘++<Identifier id>‘:
10       recordAtom("PreIncrement");
11
12     case (Expression)‘<ConditionalOrExpression e> ? <Expression e1> : <ConditionalExpression e>‘:
13       recordAtom("ConditionalOperator");
14
15     // ...
16   }
17 }
```

The `main` program

```
1  map[str, int] atoms = ();
2
3  public void execute(loc srcdir) {
4    files = findAllFiles(srcdir, "java");
5    real errors = 0.0;
6    for(f <- files) {
7      try {
8        CompilationUnit unit = parse(#CompilationUnit, f);
9        analyse(unit);
10     }
11     catch e: {
12       errors += 1;
13     }
14   }
15
16   for(k <- atoms) {
17     println("<k> : <atoms[k]>");
18   }
19
20   totalFiles = size(files);
21   println("Could not parse <errors> files (<errors * 100.0 / totalFiles> %)");
22 }
```