EASY Meta-Programming with Rascal
Leveraging the Extract-Analyze-SYNthesize Paradigm

Paul Klint & Jurgen Vinju

Joint work with (amongst others):
Bas Basten, Mark Hills, Anastasia Izmaylova, Davy Landman, Arnold Lankamp, Bert Lisser, Atze van der Ploeg, Tijs van der Storm, Vadim Zaytsev
Cast of Our Heroes

- Alice, system administrator
- Bernd, forensic investigator
- Charlotte, financial engineer
- Daniel, multi-core specialist
- Elisabeth, model-driven engineering specialist
Meet Alice

- Alice is security administrator at a large online marketplace
- **Objective**: look for security breaches
- **Solution**:
  - Extract relevant information from system log files, e.g. failed login attempts in Secure Shell
  - Extract IP address, login name, frequency, …
  - Synthesize a security report
Meet Bernd

- **Bernd**: investigator at German forensic lab
- **Objective**: finding common patterns in confiscated digital information in many different formats. This is very labor intensive.
- **Solution**:
  - Design **DERRICK** a domain-specific language for this type of investigation
  - Extract data, analyze the used data formats and synthesize Java code to do the actual investigation
Meet Charlotte

- Charlotte works at a large financial institution in Paris
- **Objective:** connect legacy software to the web
- **Solution:**
  - extract call information from the legacy code, analyze it, and synthesize an overview of the call structure
  - Use entry points in the legacy code as entry points for the web interface
  - Automate these transformations
Meet Daniel

- Daniel is concurrency researcher at one of the largest hardware manufacturers worldwide
- **Objective**: leverage the potential of multi-core processors and find concurrency errors
- **Solution**:
  - extract concurrency-related facts from the code (e.g., thread creation, locking), analyze these facts and synthesize an abstract automaton
  - Analyze this automaton with third-party verification tools
Meet Elisabeth

- Elisabeth is software architect at an airplane manufacturer
- **Objective:** Model reliability of controller software
- **Solution:**
  - describe software architecture with UML and add reliability annotations
  - Extract reliability information and synthesize input for statistics tool
  - Generate executable code that takes reliability into account
What are their **Technical Challenges**?

- How to parse source code/data files/models?
- How to extract facts from them?
- How to perform computations on these facts?
- How to generate new source code (trafo, refactor, compile)?
- How to synthesize visualizations, charts?

**EASY: Extract-Analyze-SYnthesize Paradigm**
Why a new Language?

- No current technology spans the full range of EASY steps
- There are many fine technologies but they are
  - highly specialized with steep learning curves
  - hard to learn unintegrated technologies
  - not integrated with a standard IDE
  - hard to extend

Goal
Keep all benefits of advanced (academic) tools and unify them in a new, extensible, teachable framework
Here comes Rascal to the Rescue
Rascal Elevator Pitch
Rascal Elevator Pitch

- Sophisticated built-in data types
- Immutable data
- Static safety
- Generic types
- Local type inference
- Pattern Matching
- Syntax definitions and parsing

- Concrete syntax
- Visiting/traversal
- Comprehensions
- Higher-order
- Familiar syntax
- Java and Eclipse integration
- Read-Eval-Print (REPL)
Rascal ...

- is a new language for meta-programming
- is based on Syntax Analysis, Term Rewriting, Relational Calculus
- extended super set (regarding features not syntax!) of ASF+SDF and Rscript
- relations used for sharing and merging of facts for different languages/modules
- embedded in the Eclipse IDE
- easily extensible with Java code
Rascal design based on ...

- **Principle of least surprise**
  - Familiar (Java-like) syntax
  - Imperative core
- **What you see is what you get**
  - No heuristics (or at least as few as possible)
  - *Explicit* preferred over *implicit*
- **Learnability**
  - Layered design
  - Low barrier to entry
Rascal provides

- Rich (immutable) data: lists, sets, maps, tuples, relations, ... with comprehensions and many operators
- Syntax definitions & parser generation
- Syntax trees, tree traversal
- Pattern matching (text, trees, lists, sets, ...) and pattern-directed invocation
- Code generation (string templates & trees)
- Java and Eclipse (IMP) integration
Bridging Gaps

Rascal Programming

Analysis
- Parsing/Matching
- Comprehension
- Projection
- Extraction
- Traversal

Data
- ASTs
- Sets
- relations

Synthesis
- Abstract syntax
- Concrete syntax
- Rewriting
- Annotation
- Templates

Visualization

Figure
One-stop-shop

- Cool parsers
- Deal of the day: Cheap type checkers
- Fancy visualization
- Just in: new modeling gadgets
Some Classical Examples

- Read-Eval-Print
- Hello
- Factorial
- ColoredTrees
Read-Eval-Print

```
rascal>1 + 1
int: 2

rascal>[1,2,3]
list[int]: [1,2,3]

rascal>[1,2,3] + [9,5,1]
list[int]: [1,2,3,9,5,1]
```

List concatenation
Read-Eval-Print

\[
\text{rascal}\>\{1,2,3\} \\
\text{set}[\text{int}]: \{1,2,3\}
\]

\[
\text{rascal}\>\{1,2,1\} \\
\text{set}[\text{int}]: \{1,2\}
\]

\[
\text{rascal}\>\{1,2,3\} + \{9,5,1\} \\
\text{set}[\text{int}]: \{1,2,3,9,5\}
\]

Sets do not contain duplicates

Set union
Read-Eval-Print

\texttt{rascal}\{i*i | i \leftarrow [1..10]\}
\texttt{set[int]}: \{1,4,9,16,25,36,...\}

\texttt{rascal}\{i*i | i \leftarrow [1..10], t \% 2 == 0\}
\texttt{set[int]}: \{4,16,36,...\}
rascal> import IO;
ok
rascal> for (i <- [1..10]) {
    >>>>>>>>> println("<i> * <i> = <i * i>");
    >>>>>>>>>}
1 * 1 = 1
2 * 2 = 4
3 * 3 = 9
4 * 4 = 16
5 * 5 = 25
6 * 6 = 36
7 * 7 = 49
8 * 8 = 64
9 * 9 = 81
10 * 10 = 100
list[void]: [ ]
Hello (on the command line)

```rascal
rascal > import IO;
ok

rascal> println("Hello, my first Rascal program");
Hello, my first Rascal program

ok
```
Hello (as function in module)

```rascal
module demo::basic::Hello
import IO;
public void hello() {
    println("Hello, my first Rascal program");
}
```

```rascal
rascal> import demo::basic::Hello;
ok

rascal> hello();
Hello, my first Rascal program
ok
```
## Factorial

module demo::Factorial
public int fac(int N){
    return N <= 0 ? 1 : N * fac(N - 1);
}

rascal> import demo::Factorial;
ok

rascal> fac(47);
fac(47):
ing: 2586232415111681806429643551536119799691976323891200000000000
Types and Values

- **Atomic**: bool, num, int, real, str, loc (source code location), datetime
- **Structured**: list, set, map, tuple, rel (n-ary relation), abstract data type, parse tree
- **Type system**:
  - Types can be parameterized (polymorphism)
  - All function signatures are explicitly typed
  - Inside function bodies types can be inferred (local type inference)
<table>
<thead>
<tr>
<th>Type</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>true, false</td>
</tr>
<tr>
<td>int, real</td>
<td>1, 0, -1, 123, 1.023e20, -25.5</td>
</tr>
<tr>
<td>str</td>
<td>“abc”, “values is &lt;x&gt;”</td>
</tr>
<tr>
<td>loc</td>
<td></td>
</tr>
<tr>
<td>datetime</td>
<td>$2010-07-15T09:15:23.123+03:00</td>
</tr>
<tr>
<td>tuple</td>
<td>&lt;1,2&gt;, &lt;“john”, 43, true&gt;</td>
</tr>
<tr>
<td>list</td>
<td>[], [1], [1,2,3], [true, 2, “abc”]</td>
</tr>
<tr>
<td>set</td>
<td>{}, {1,3,5,7}, {“john”, 4.0}</td>
</tr>
<tr>
<td>rel</td>
<td>{&lt;1,10,100&gt;,&lt;2,20,200&gt;}</td>
</tr>
<tr>
<td>map</td>
<td>(), (“a”:1, “b”:2,”c”:3)</td>
</tr>
<tr>
<td>node</td>
<td>f, add(x,y), g(“abc”,[2,3,4])</td>
</tr>
</tbody>
</table>
User-defined datastructures

- Named alternatives
  - name acts as constructor
  - can be used in patterns
- Named fields (access/update via . notation)
- All datastructures are a subtype of the standard type node
  - Permits very generic operations on data
- Parse trees resulting from parsing source code are represented by the datatype Tree
ColoredTrees: CTree

data CTree = leaf(int N)  
            | red(CTree left, CTree right)  
            | black(CTree left, CTree right) ;

rb = red(black(leaf(1), red(leaf(2), leaf(3))),  
        black(leaf(4), leaf(5)));
Abstract Syntax

data STAT = asgStat(Id name, EXP exp)
    | ifStat(EXP exp,list[STAT] thenpart, list[STAT] elsepart)
    | whileStat(EXP exp, list[STAT] body)
;
Type Hierarchy

- **bool**
- **int**
- **real**
- **str**
- **loc**
- **list**
- **map**
- **tuple**
- **set**
- **map**
- **rel**
- **node**

- **void**

**ADT**

- **A\textsubscript{1}**
- **...**
- **A\textsubscript{n}**

**alias**

- **ADT\textsubscript{1}**
- **...**
- **ADT\textsubscript{n}**

**data**

- **Tree**

**EASY**

- **C**
- **Java**
- **...**

- **= subtype-of**
Pattern matching

Given a pattern and a value:

- Determine whether the pattern matches the value
- If so, bind any variables occurring in the pattern to corresponding subparts of the value
Pattern matching

Pattern matching is used in:

- Explicit **match operator** `Pattern := Value`
- **Switch**: matching controls case selection
- **Visit**: matching controls visit of tree nodes

* **Function calls**: matching controls dynamic dispatch
Patterns

Regular: Grep/Perl like regular expressions

/^<before:\W*><word:\w+><after:.*$>/

Abstract: match data types

whileStat(Exp, Stats*)

Concrete: match parse trees

`while <Exp> do <Stats*> od`
Regular Patterns

```
rascal>/^[a-z]+/ := "abc"
bool: true

rascal>/rac/ := "abracadabra";
bool: true

rascal>/^[rac]/ := "abracadabra";
bool: false

rascal>/rac$/ := "abracadabra";
bool: false
```
Regular Patterns

```rascal
rascal>if(/\W<x:\[a-z]+>/ := "12abc34")
   println("x = <x>");
ok
```

- Matches non-word characters ($\W$) followed by one or more letters.
- Binds text matched by $[a-z]+$ to variable $x$. (Is only available in the body of the if statement)
- Prints: abc.
- Regular patterns are tricky (in any language)!
Patterns

Abstract/Concrete patterns support:

- List matching: \([ P_1, ..., P_n]\)
- Set matching: \(\{P_1, ..., P_n\}\)
- Named subpatterns: \(N:P\)
- Anti-patterns: \(!P\)
- Descendant: \(/N\)

Can be combined/nested in arbitrary ways
List Matching

\[
\text{rascal}\> L = [1, 2, 3, 1, 2]; \\
\text{list[int]}: [1, 2, 3, 1, 2]
\]

\[
\text{rascal}\> [X^*, 3, X] := L; \\
\text{bool}: \text{true}
\]

\[
\text{rascal}\> X; \\
\text{Error: \text{X is undefined}}
\]

\[
\text{rascal}\> \text{if}([X^*, 3, X] := L) \text{println("X = \text{<X>"});} \\
X = [1, 2] \\
\text{ok}
\]

List pattern

\(X^*\) is a list variable and abbreviates \(\text{list[int]} \ X\)

\(X\) is bound but has limited scope

List matching provides \textbf{associative (A) matching}
Set Matching

rascal> S = {1, 2, 3, 4, 5};
set[int]: {1,2,3,4,5}

rascal> {3, Y*} := S;
bool: true

rascal> if({3, Y*} := S) println("Y = \<\text{Y}\>");
Y = {5,4,2,1}
ok

Set pattern

\(Y^*\) is a set variable and abbreviates \(\text{set}[\text{int}]\ Y\)

Set matching provides associative, commutative, identity (ACI) matching
Note

- List and Set matching are non-unitary
- E.g., \([L^*, M^*] := [1, 2]\) has three solutions:
  - \(L == [], M == [1,2]\)
  - \(L == [1], M == [2]\)
  - \(L == [1,2], M == []\)
- In boolean expressions, matching, etc. solutions are generated when failure occurs later on (local backtracking)
Descendant Matching

whileStat(_, /ifStat(_, _, _))

Match a while statement that contains an if statement at arbitrary depth
Enumerators and Tests

- Enumerate the elements in a value
- Tests determine properties of a value
- Enumerators and tests are used in comprehensions

* And in if, for, while, etc.
Enumerators

- Elements of a list or set
- The tuples in a relation
- The key/value pairs in a map
- The elements in a datastructure (in various orders!)

```
int x <- { 1, 3, 5, 7, 11 }
int x <- [ 1 .. 10 ]
asgStat(Id name, _) <- P
```
Comprehensions

- Comprehensions for lists, sets and maps
- Enumerators generate values; tests filter them

\[
\text{rascal}\triangleright \{n * n \mid \text{int } n \leftarrow [1 \ldots 10], \ n \ % \ 3 \ == \ 0\}; \\
\text{set[\text{int}]}: \{9, 36, 81\}
\]

\[
\text{rascal}\triangleright [ \ n \mid /\text{leaf(\text{int } n) } \leftarrow \text{rb} ]; \\
\text{list[\text{int}]}: \{1,2,3,4,5\}
\]

\[
\text{rascal}\triangleright \{\text{name} \mid /\text{asgStat(id name, _)} \leftarrow P\}; \\
\{\ldots\}
\]
Control structures

- Combinations of enumerators and tests drive the control structures
- `for`, `while`, `all`, `one`

```
rascal> for(/int n ← rb, n > 3){ println(n);}
4
5
ok
rascal> for(/asgStat(Id name, _) ← P, size(name)>10){
    println(name);
}
...
```
Counting words in a string

```rascal
public int countWords(str S){
    int count = 0;
    for(/[a-zA-Z0-9]+/ := S){
        count += 1;
    }
    return count;
}
```

```rascal
countWords( "'Twas brillig, and the slithy toves" ) => 6
```
Switching

- A **switch** does a top-level case distinction

```rascal
switch (P){
    case whileStat(EXP Exp, Stats*):
        println("A while statement");
    case ifStat(Exp, Stats1*, Stat2*):
        println("An if statement");
}
```

Every switch is a code smell, you can also use functions dispatched by patterns
Enough!

- Ok, that was quite a lot of information
- Rascal is for Meta-Programming
  - Code analysis
  - Code transformation
  - Code generation
  - Code visualization
- It is a normal programming language
- Learn it using the Tutor view and the Console