EASY Programming with Rascal

Paul Klint

Joint work with
Emilie Balland, Bas Basten, Jeroen van den Bos, Mark Hills, Arnold Lankamp, Bert Lisser, Tijs van der Storm, Jurgen Vinju

Opening BLDL, November 4, 2009, Bergen, Norway
Cast of 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
What are their Common Problems?

- How to parse source code/data files
- How to extract facts from it
- How to perform computations on these facts
- How to generate new source code
- How to synthesize other information

EASY: Extract-Analyze-SYNthesize Paradigm
System Under Investigation (SUI)

EASY Paradigm

Extract

Internal Representation

Analyze

Synthesize

Results
What tools are available to our heroes?

- **Lexical tools**: Grep, Awk, Perl, Python, Ruby
  - Regular expressions have limited expressivity
  - Hard to maintain
- **Compiler tools**: yacc, bison, CUP, ANTLR
  - Only automate front-end part
  - Everything else programmed in C, Java, ..
- **Attribute Grammar tools**: FNC2, JastAdd, …
  - Mostly analysis, weak in transformation
What tools are available to our heroes?

- **Relational Analysis tools**: Grok, Rscript
  - Strong in analysis
- **Transformation tools**: ASF+SDF, Stratego, TOM, TXL
  - Strong in transformation
- **Logic languages**: Prolog
- Many others …

Apologies if your favorite tool does not appear in this list
<table>
<thead>
<tr>
<th></th>
<th>Extract</th>
<th>Analyze</th>
<th>Synthesize</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Lexical tools</strong></td>
<td>++</td>
<td>+/-</td>
<td>--</td>
</tr>
<tr>
<td><strong>Compiler tools</strong></td>
<td>++</td>
<td>+/-</td>
<td>+/-</td>
</tr>
<tr>
<td><strong>Attribute grammar tools</strong></td>
<td>++</td>
<td>+/-</td>
<td>--</td>
</tr>
<tr>
<td><strong>Relational tools</strong></td>
<td>--</td>
<td>++</td>
<td>--</td>
</tr>
<tr>
<td><strong>Transformation tools</strong></td>
<td>--</td>
<td>+/-</td>
<td>++</td>
</tr>
<tr>
<td><strong>Rascal</strong></td>
<td>++</td>
<td>++</td>
<td>++</td>
</tr>
</tbody>
</table>
Why a new Language?

• No current technology spans the full range of EASY steps

• There are many fine technologies but they are
  • highly specialized
  • hard to learn
  • not integrated with a standard IDE
  • hard to extend
  • ...
Here comes Rascal to the Rescue
Rascal ...

• ... is a new language for meta-programming
• ... supports the EASY paradigm
• ... is based on
  • Syntax Analysis
  • Term Rewriting
  • Relational Calculus
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)
HOW EXCITING!!
PLEASE TELL ME MORE!!
A Domain-specific Language for State Machines
canReach(
  finite-state machine
  state S1;
  state S2;
  state S3;
  trans a: S1 -> S2;
  trans b: S2 -> S1;
  trans a: S2 -> S3
)

= (S1: {S1, S2, S3},
   S2: {S1, S2, S3},
   S3: {}})
State Machine Concrete Syntax

module demo/StateMachine/Syntax

... 

"state" Id -> State
"trans" Id ":" Id ":->" Id -> Trans
State -> Decl
Trans -> Decl
"finite-state" "machine" {Decl ":"}+ -> FSM
module demo::StateMachine::CanReach

import demo::StateMachine::Syntax;
import Graph;

FSM example =

  finite-state machine
  
  state S1;
  state S2;
  state S3;
  trans a: S1 -> S2;
  trans b: S2 -> S1;
  trans a: S2 -> S3;

... (next sheet)
CanReach (2)

```rascal
module demo::StateMachine::CanReach
... (previous sheet)

public map[str, set[str]] canReach(FSM fsm){
    transitions = getTransitions(fsm);
    closure = transitions+;
    return ( s : closure[s] | str s <- carrier(transitions) );
}
```

Extract transitions as a graph
Transitive closure
Map comprehension
Enumerate all states

return a map in which each state is associated with all states that can be reached from it
module demo::StateMachine::CanReach
...

public graph[str] getTransitions(FSM fsm){
    return {
        { "<from>", "<to>" |
          /* trans <Id a>: <Id from> -> <Id to> ` <- fsm */
    }
}

Enumerate all transitions in the FSM
Convert a tree element to a string
Concrete pattern with variables
Example

Generating Getters and Setters
Generating Getters and Setters

• Given:
  • A class name
  • A mapping from names to types

Required:
• Generate the named class with getters and setters
Generating getters and setters: Input

public map[map] fields = (  
  "name" : "String",  
  "age" : "Integer",  
  "address" : "String"  
);

Field name of type String
Field age of type Integer
Field address of type String

genClass("Person", fields)
Generate class person with these fields
Generating getters and setters

Expect Output

```java
public class Person {
    private Integer age;
    public void setAge(Integer age) { this.age = age; }
    public Integer getAge() { return age; }

    private String name;
    public void setName(String name) { this.name = name; }
    public String getName() { return name; }

    private String address;
    public void setAddress(String address) { this.address = address; }
    public String getAddress() { return address; }
}
```
Generating Getters and Setters

```java
public str genClass(str name, map[str,str] fields) {
    return "public class <name> {
        <for (f <- fields) {
            str t = fields[f];
            str n = capitalize(f);>
            private <t> <f>;
            public void set<n>(<t> <f>) { this.<f> = <f>; } 
            public <t> get<n>() { return <f>; } 
        }
    }
};"
}
```
Fact extraction and visualization
While working on a Java project ... 

- For example, jspwiki
- What are the different file types used in this project?
What are the file types in this project?

module demo::filetypes
import Resources;
import viz::Chart;

public void main(){
    jspwiki = getProject(|project://jspwiki|);
    extensions = ();
    visit(jspwiki){
        case file(loc l): extensions[l.extension]? 0 += 1;
    }
    pieChart("Extensions", extensions, dim3());
}
Extensions

EASY Meta-Programming with RASCAL
The Rascal Standard Library

- Benchmark
- Boolean
- Exception
- (Labelled) Graph
- Integer
- IO
- JDT (Eclipse only)
- List
- Location
- Map
- Node

- Real
- Relation
- RSF
- Resource (Eclipse only)
- Set
- String
- Subversion
- Tuple
- ValueIO
- viz::Chart
- viz::View (Eclipse only)
Long-term Perspective

• The Rascal language supports the EASY paradigm:
  • creation and execution of fact analysis and transformation tools
  • DSLs
  • meta-programming
• Familiar notation and Eclipse integration lower barrier to entry
• Work in progress
Information

General information:
http://www.meta-environment.org

Latest version of Rascal documentation:

Download Rascal implementation:
http://www.meta-environment.org/Meta-Environment/Rascal
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