Scripting a Refactoring with Rascal and Eclipse

Mark Hills, Paul Klint, & Jurgen J. Vinju

Fifth Workshop on Refactoring Tools 2012
June 1, 2012
Rapperswil, Switzerland

http://www.rascal-mpl.org
Overview

• A Brief Introduction to Rascal

• The Visitor to Interpreter Refactoring

• Extending to Other Languages and Refactorings

• Related Work
Overview

- A Brief Introduction to Rascal
- The Visitor to Interpreter Refactoring
- Extending to Other Languages and Refactorings
- Related Work
What is Rascal?

Rascal is a powerful domain-specific programming language that can scale up to handle challenging problems in the domains of:

- Software analysis
- Software transformation
- DSL Design and Implementation
Rascal Goals

• Cover entire domain of meta-programming

• "No Magic" -- users should be able to understand what is going on from looking at the code

• Programs should look familiar to practitioners

• Unofficial “language levels” -- users should be able to start simple, build up to more advanced features
Rascal Features

• Scannerless GLL parsing

• Flexible pattern matching, lexical backtracking, and matching on concrete syntax

• Functions with parameter-based dispatch, default functions, and higher-order functions

• Traversal and fixpoint computation operations

• Immutable data, rich built-in data types, user-defined types

• Rich collection of libraries
Overview

• A Brief Introduction to Rascal

• The Visitor to Interpreter Refactoring

• Extending to Other Languages and Refactorings

• Related Work
Visitor to Interpreter: Motivation

- Developed as part of an experiment in software maintenance
- Question: maintenance cost of visitor versus interpreter
- Goal: two systems, with only this variable
- Solution: build a refactoring!
V2I, From 30,000 Feet

1. Extract facts needed for transformation
2. Do preparatory transformations
3. Generate interpreter code
4. Clean up
public Result<IValue> visitExpressionFieldUpdate(FieldUpdate x) {
    Result<IValue> expr = x.getExpression().accept(this);
    Result<IValue> repl = x.getReplacement().accept(this);
    String name = Names.name(x.getKey());
    return expr.fieldUpdate(name, repl, getCurrentEnvt().getStore());
}

public Result<IValue> interpret(Evaluator __eval) {
    Result<IValue> expr = this.getExpression().interpret(__eval);
    Result<IValue> repl = this.getReplacement().interpret(__eval);
    String name = org.rascalmpl.interpreter.utils.Names.name(this.getKey());
    return expr.fieldUpdate(name, repl, __eval.getCurrentEnvt().getStore());
}
Extract Facts Needed for Transformation

- Extract JDT Facts
- Calculate extends and inherits for visitor interface
- Find all visit method implementations
- Find all non-public field and method dependencies
Do Preparatory Transformations

- Run code cleanup on implementers
- Make non-public dependencies public
- Fully qualify type names
Generate Interpreter Code

- Transform visit methods to interpret methods using string matching/replacement
- Generate new interpreter class hierarchy for new interpret methods
Why Not Move Method and Push Down?

- Still need to do much preparatory work
- Less control (e.g., public fields versus getters and setters, no copy method)
- Still need to transform method bodies
- Can produce broken code
Clean Up

- Perform clean up on generated code, including adding imports
- Remove old visit methods
Overview

- A Brief Introduction to Rascal
- The Visitor to Interpreter Refactoring
- Extending to Other Languages and Refactorings
- Related Work
Will This Work Elsewhere?

- Makes heavy use of JDT, Eclipse refactoring API

- Technique isn’t Java specific, should work for other language given similar infrastructure

- Technique isn’t Eclipse specific, Rascal just happens to work best with Eclipse

- Using a different IDE would require bridging software (e.g., something to talk to Emacs, NetBeans, etc)

- Overall: easier to change language, harder to change IDE
Overview

• A Brief Introduction to Rascal

• The Visitor to Interpreter Refactoring

• Extending to Other Languages and Refactorings

• Related Work
Related Work

- Rascal: Infer Generic Type Arguments with FJ, TyMoRe (Anastasia)

- JastAdd-based refactorings

- Languages for refactorings: Refacola, JunGL, DSL in Wrangler
For More Information on Rascal: http://tutor.rascal-mpl.org

Welcome to RascalTutor

RascalTutor is an interactive learning environment to learn and practice Rascal-related concepts. It is a work in progress, currently most information is organized as a browsable manual.

Courses for Rascal Users

First read this

- EASY: The Extract-Analyze-SYthesize paradigm,
- SolutionStrategies: How to arrive at a problem solution,
- Rascal Compared With Other Paradigms,

... and then this

- Rascal: Language and libraries,
- Recipes: step-by-step solutions to common tasks,
- Online Rascal Tests.

Courses for Rascal Contributors

- Tutor: Writing Courses using the RascalTutor,
- Testing the Tutor (internal use only).
• Rascal: http://www.rascal-mpl.org

• SEN1: http://www.cwi.nl/sen1

• Me: http://www.cwi.nl/~hills
Related Work: Refactoring with Meta-Programming Languages

• M. Schäfer, T. Ekman, and O. de Moor. Sound and Extensible Renaming for Java (OOPSLA’08)

• M. Schäfer, M. Verbaere, T. Ekman, and O. de Moor. Stepping Stones over the Refactoring Rubicon (ECOOP’09)

• M. Schäfer and O. de Moor. Specifying and Implementing Refactorings (OOPSLA’10)
Related Work: Refactoring using Rascal

- P. Klint, T. van der Storm, and J. J. Vinju. RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation (SCAM’09)

- TyMoRe: **Type** based **Modular Refactorings**, i.e., refactorings using type constraints, with a specific focus on reuse
Related Work: Scripting Refactorings/Refactoring DSLs


- **JunGL**: M. Verbaere, R. Ettinger, and O. de Moor. JunGL: A Scripting Language for Refactoring (ICSE’06)

- **Wrangler**: H. Li and S. J. Thompson. A Domain-Specific Language for Scripting Refactorings in Erlang (FASE’12)