Rascal Tutorial

Versions of this document:

- HTML: http://www.cwi.nl/∼storm/rascal-tutorial (this page)
- PDF: http://www.cwi.nl/∼storm/rascal-tutorial/handout.pdf

The slides for this tutorial can be found online at:

- Introduction intro.pdf
- Syntax and transformation 1-syntax-transformation.pdf
- Extraction and analysis 2-extraction-analysis.pdf
- Code generation 3-codegen.pdf

The archive with a project containing solutions to the exercises can be found here miss-grant-full.zip

Comments and suggestions: Tijs van der Storm storm@cwi.nl

Introduction

Rascal is programming language for source code analysis and transformation. The primary application areas are (legacy) system renovation, reverse engineering and re-engineering, and the implementation of domain specific languages (DSLs). DSLs are languages tailored to a specific application domain. Examples include SQL, Excel, Make, Latex, VHDL, etc. Today we are going to use Rascal for some aspects of implementing a small domain specific language for state machines. The example is derived from Martin Fowler's book on Domain Specific Languages. The relevant chapter is published on-line:

> http://www.informit.com/articles/article.aspx?p=1592379

State machines are useful for describing state dependent behaviour, for instance to control machines or work-flow engines. For more information on the theory of this kind of state machines, you may want to consult Wikipedia http://en.wikipedia.org/wiki/Finite_state_transducer.

In this tutorial we will explore the Rascal language and environment by encountering the following aspects of DSL implementation:

- A context-free grammar to describe the syntax of the state machine language
- An algebraic data type (ADT) for describing state machine abstract syntax trees (ASTs)
- Reset events (see the link above) are syntactic sugar: they can be *desugared* into an equivalent state machine that does not use them.
- Extraction of relations from a state machine. This allows easier analysis of state machines. The relations can be connected to the state machine visualizer (provided by us).
- A consistency checker for state machines. This component, for instance, highlights use of undefined states or events, marks duplicate states, commands or events and detects unreachable states.
- A code generator that produces a Java code consuming and producing tokens.

- A transformation that takes two state machines and produces a new state machine that runs the two original machines in parallel. The resulting state machine can be input to the original code generator and visualizer.
- Provide domain-specific IDE features for state machines: context-menus to invoke the code generator, outline views, folding, error marking.

Installation

If you haven't installed the Rascal Eclipse Plugin already, go to the following URL and follow the instructions.

- http://www.rascal-mpl.org/Rascal/EclipseUpdate

Some notes:

- You need to install *Eclipse for RCP and RAP Developers* (Indigo) if you haven't already.
- You have to have a Java SDK available (JRE is not enough).
- Make sure you use 64 bit version of Eclipse if your JVM is 64 bit (mutatis mutandis for 32 bit).
- Project names with spaces or installing in a directory with spaces will not work.

Download the zip archive for the tutorial.

- http://www.cwi.nl/∼storm/rascal-tutorial/miss-grant.zip

To start, open the Rascal perspective and "Import..." the Eclipse project contained in the zip file (select the archive file).

To open a Rascal console, open a Rascal file from the project and right-click on the Rascal editor. In the context-menu you'll find Launch Console.

Rascal has an interactive documentation system which can started from within Eclipse via the Rascal menu. It can be found online at http://tutor.rascal-mpl.org.

Structure of the project

The source directory of project contains a number of files. Some of them have real code in them, other have stubs that you will implement during this tutorial. The files are briefly described below:

- AST.rsc: this contains an ADT that defines the abstract syntax of state machines.
- Check.rsc: a consistency checker for state machines. To be implemented.
- Compile.rsc: stubs for a code generator to Java.
- Desugar.rsc: stubs for desugaring reset events.
- Extract.rsc: fact extraction from state machines. To be implemented.
- Implode.rsc: helper functions to obtain ASTs from parse trees.

- Merge.rsc: template for merging two state machines to obtain another state machine that executes the original two in parallel. Bonus exercise.
- Parse.rsc: helper functions to parse files and strings.
- Plugin.rsc: the top-level project module. This hooks up various functionality to the Eclipse IDE for state machines.
- Syntax.rsc: the context-free grammar of state machines. Check it out if you want to learn about Rascal's powerful grammar formalism.
- Unparse.rsc: stubs for a "pretty" printer of state machines and some tests that assert that it works correctly. To be implemented.
- Visualize.rsc: functions to graphically visualize a state machine. Check it out if you want to learn more about Rascal's visualization library.

The special folders eclipse and std contain Rascal's standard library. Have a peek to see what's there. Finally, the input directory contains some example state machine definitions. If you've opened a console and imported the Plugin module (import Plugin;) you may invoke the main() function. If you then double-click controller files an editor will open with syntax highlighting.

Syntax Definition for State Machines

Rascal has built-in support for context-free grammars which can be used to define the syntax of programming languages. A syntax rule consists of the following parts:

```
syntax NonTerminal = label_1: Element_1i ... Element_1m1
                   | ...
                   | label_n: Element ... Element_nmn ;
```

This defines a rule named "NonTerminal" with *n* alternatives. Each alternative has a label and a sequence of Elements. The elements of an alternative define the syntax to be recognized by this rule. An element can be one of the following symbols:

- "a literal"
- a NonTerminal
- a regular symbol: X? for optional, X* for zero-or-more, X+ for one-or-more, and the separated list operators: {X "sep"}*, and {X "sep"}+. X can be any symbol, but typically will be a non-terminal

To define lexical rules (e.g., for identifiers) character classes are used (similar to regular expressions):

- char class [a-z]: recognize a character between a and z. Or: [\t\n\r\ ]: recognize a white-space character
- ![a-z]: recognize any character that is not a lowercase alphabetic character
- ?, *, and + can be used on character classes as well

The Syntax module of the project already defines the syntax of Fowler's state machines. Note that our syntax definition extends the Layout module from the Rascal standard library. Have a look and try to understand the how Rascal's syntax formalism works.

Tip: open the std entry in the package explorer of the MissGrant project to see what's in the standard library. Locate and open the Layout module and digest what's in there.

Quiz: what is the layout convention defined in the Comment module?

Quiz: what is the meaning of the !<< and !>> constructs? Why are they needed?

If you've run main() from the Plugin module, you can already open editors with state machines. If you want to parse state machines from the console, import module Parse, and use the provided parse functions.

If you import Implode, you can use the load function to directly obtain ASTs for a source location. You can inspect parse trees and ASTs by importing util::ValueUI, and entering tree(*the tree*) in the console.

Quiz: what is the meaning of # in the definition of parse? Hint: type in "#int" in the Rascal console.

Abstract Syntax

Rascal uses algebraic data types (ADTs) for describing abstract syntax, as is common in functional programming languages. The Rascal standard library defines a function "implode" that turns a concrete syntax tree into an abstract syntax tree.

To make this work, an abstract syntax ADT should correspond to the syntax definition in the following way:

- Every non-terminal maps to an ADT type: for non-terminal X, define "data X ="
- Every alternative of a non-terminal X maps to a constructor alternative of the ADT X, where the name of the constructor must correspond to the label of alternative.
- Every element in an alternative that is not a literal, maps to a constructor argument.
- Regular symbols X?, X*, X+, {X "sep"}*, {X "sep"}+ map to list[T] where T is the type corresponding to X.
- Lexical symbols (e.g., identifiers) map to the int, str, real or bool data types.

You can verify in the AST module that the abstract syntax definition indeed follows this scheme.

Quiz: what are source code transformations where ASTs are cumbersome to use?

Exercise 1: source to source transformation

Consider the following paragraph in Fowler's text:

> In particular, you should note that reset events aren't strictly necessary to express Miss Grant's controller. As an alternative, I could just add a transition to every state, triggered by doorOpened, leading to the idle state. The notion of a reset event is useful because it simplifies the diagram.

What this means is, that it is possible to construct an equivalent state machine that does not depend on reset events. In other words, the resetEvents feature of the state machine language is *syntactic sugar*. In this assignment you are to write a transformation that *desugars* reset events according to the quote above.

Tip: use the Rascal visit construct to transform the state machine AST.

Bonus: implement the reverse, i.e. 'resugaring' of resetEvents.

Quiz: does resugaring after desugaring always give the original state machine? If not, why not, and would that be a problem?

Exercise 2: Fact extraction

For some applications, especially source code analyses, the tree structure of the AST is not ideal. In this assignment you will write a function that extracts a relational abstraction of state machines. Relations are natural representations for graphs. And a state machine can be considered as a special kind of graph.

In order to connect the resulting analysis to the state machine visualizer, we use the following types as interfaces (see module Extract):

```
alias TransRel = rel[str state, str event, str toState]
alias ActionRel = rel[str state, str command]
```

The first relation captures the transition structure of a state machine: it contains tuples <s, t, s'>, where s is the source state, t the triggering event, and s' the target state. The second relation captures which commands should be executed upon entering a certain state.

NB: You may assume that reset events have been desugared as in the previous step.

Quiz: what is the difference between a binary relation and a map?

Exercise (optional): implement the extraction of two similar relations, only now the produce a transition relation between state, event *token* and state, and an action relation between state and command *token*.

Visualizing a state machine

The relations of the previous assignment are an abstract representation of a state machine. This provides excellent input to graphically visualizing a state machine. The provided Rascal project contains a module to visualize state machines (Visualize). If you've implemented the relations of the previous exercise correctly, you can visualize a state machine clicking MissGrant -> Visualize in the context menu of a state machine editor. You can also import the module Visualize in the console and then call the function renderController on a controller AST.

In the visualization there are buttons for each event. You can click on them to interactively simulate the execution of the state machine.

Exercise 3: Well-formedness checking of state machines

Many programming languages have type checkers. In the case of state machines, there isn't really a notion of types. Nevertheless, it is still possible to make mistakes. In this assignment the goal is to make a checker function that detects such mistakes. This function will return a collection of error or warning messages. The data type to be used for this can be found in the standard library module Message.

The list of things you could check for includes (but might not be limited by) the following:

- Duplicate definitions of events/commands and their tokens.
- Duplicate state definitions.
- Reset events that are used in a transition.

- Non-determinism (two or more transitions from the same state that fire on the same token).
- Undeclared reset events, actions, events or states.
- Unreachable states.
- Unused commands or events.

Rascal checkers typically return a set of error messages which can be used by the IDE to do error marking in editors. The Message data type can be found in the Message module of the standard library. To obtain the source location of AST nodes use aNode@location (see the AST module to see which nodes have these annotations).

Exercise: write a checker for state machines in the Check module. The toplevel function is provided and will be used by the IDE.

Tip: write one helper function for each violation (e.g., duplicate events) and call each helper function in the top-level check function, unioning the respective sets of error messages.

Tip: for reachability analysis use the built-in Rascal operator for transitive closure (post-fix +).

Note: you may put the facts you've extracted from the previous assignment to good use in some of the analyses that the checker performs.

Exercise 4: Unparsing using string templates.

In this assignment you are to write a function that converts ASTs back to source code using Rascal's built-in string templates. String templates are string literals with advanced mechanisms for interpolation:

- Expression interpolation: "Hello <name>!"
- For loop interpolation: "abc<for (x <- ["c","d","e"]) {><x><}>fgh"
- If statement interpolation: "abc<if (x > 0) {><x><} else {><x + 1><}>"

The Unparse module shows function definition stubs in the pattern-based invocation style: each AST node has a corresponding "equation". Fill in each definition with the appropriate string templates and recurse on unparse where needed.

Note: the result does not have to be pretty. But the provided tests at the top of Unparse should succeed. You can run them by entering ":test" on the command line after importing Unparse.

Tip: you may use the ' (single quote) character to add artifical margins in strings. String templates will be expanded relative to the margin. This improves readability of the templates.

Quiz: why wouldn't you want to use this unparse function as a real pretty printer?

Exercise 5: Code-generation to Java

State machines have to be executed in code somehow. One approach is to generate Java code. There are multiple ways for generating code from a state machine. You may consider one of the following alternatives:

- Generate a single switch statement, which dispatches on integer constants defined for each state. Upon transitioning, a current-state variable is updated.

- Generate methods for each state which call other state methods upon transitioning.
- Generate object instantiations according to Fowler's text.

Note that upon entering states the actions (if any) should be executed. To be able to run the code, assume the input stream is a java.util.Scanner object, and use nextLine to obtain the next token. Actions print tokens onto an output stream, which can be a java.io.PrintWriter.

Quiz: what is wrong with the second approach? What is the reason that it's a problem in Java, but not, say, in Scheme?

Quiz: what are the draw-backs of using strings for code generation? What are alternatives?

Bonus: parallel merge of two state machines

The desugaring reset events (cf. above) is an instance of a simple source-to-source transformation. In this assignment we will engage in a source-to-source transformation that is slightly more complex. The goal is to take two state machines and produce a new one that runs the two original state machines in parallel.

In the realm of Gothic security systems the company noticed that sometimes a client wanted two or more hidden compartments in the same room. the two controllers may share events, such as opening the drawer. In this case it is off course more cost effective to install a single controller instead of two. This is achieved by merging the two state machines through the algorithm you are going to program now and installing the resulting state machine onto a single controller.

The states in the machine resulting from merging machines S1 and S2 are identified by tuples of the states of both machines. Execution thus starts in a the initial state <s0, u0> where s0 and u0 are the initial states of S1 and S2 respectively. Running S1 and S2 in parallel then entails the following:

- If in state <s, u>, on event e, both S1 and S2 have transitions to s', and u', the combined machine transitions to <s', u'>.
- If in state <s, u>, on event e, only S1 has a transition to s', the combined machine transitions to <s', u>.
- If in state <s, u>, on event e, only S2 has a transition to u', the combined machine transitions to <s, u'>.

Note: you have to decide how commands, events and reset events are combined and how, upon entering a combined state <s', u'>, the actions of both s' and u' are combined.

Since the result of the parallel merge transformation is again just an ordinary state machine, you can reuse the code generator of the previous assignment *as is* to run two state machines in parallel.

Exercise (optional): write an interpreter that evaluates two state machines in parallel, without performing the parallel merge.

Exercise (optional): implement other operations on state machines (e.g., concat, compose, union etc.—see the Wikipedia page cited above).

Tip: use the unparser of exercise 4 to inspect the result of the parallel merge as source code in an editor as follows: first import util::ValueUI, then call text(unparse(*some controller*)).

Closing

We have seen a number of features of Rascal for source code analysis and transformation. These features were illustrated with a simple DSL for state machines. This is not all of Rascal, however. Topics that we didn't cover today include:

- Concrete syntax patterns: this is needed to implement source-to-source transformations where preservation of comments and whitespace is essential, such as with refactorings. Instead of matching on abstract patterns (in prefix-notation) one can use the concrete syntax of your language directly in Rascal code to match parse trees.
- More IDE features: hyperlinking of identifiers to their declarations and outliners. Work on completion (content-assist) is in progress.
- Visualization: Rascal includes an elaborate subsystem for declarative programming of interactive, animated visualizations.
- Java analysis: Rascal provides a bridge to the Eclipse JDT to analyse Java projects. See lang::java::jdt::JDT in the eclipse special folder.
- Additional data types: rationals, date time, XML nodes etc.
- Modular language development: apart from module import, Rascal features module extension. Together with pattern-based dispatch this enables modular development of DSLs.
- Ambiguity checking: although in theory ambiguity is undecidable, it is possible to provide practical feedback to the user about whether a grammar might be ambiguous. Check out the context-menu of Rascal editors to run *AmbiDexter* on your grammar.
- Mining source code repositories: work is in progress to provide a library to access source code repositories (Git, Subversion, CVS, etc.). This allows the analysis of version histories and evolution.

For more information, check out the online documentation http://tutor.rascal-mpl.org. If you have questions, feel free to ask them on the Rascal StackOverflow site http://ask.rascal-mpl.org.