Five Languages Symposium: Rascal

This document can also be viewed on-line. PDF can be found here.

Authors: Tijs van der Storm and Atze van der Ploeg.

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 to implement 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. DSLs are a particular form of model-driven development (MDE), where software is specified using high-level models, from which the implementation is generated.

For more information on the theory of this kind of state machines, you may want to consult Wikipedia.

In the tutorial we will explore the Rascal language and environment by implementing the following facets of the DSL:

- 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. (This is called a Model-to-Text transformation)
- A transformation that takes two state machines and produces a new state machine that runs the two original machines in parallel. (This is called a Model-to-Model transformation). 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.
- Bonus: a source-to-source transformation to implement a rename refactoring for states, events and/or commands.

Some of these assignments are more complicated than others. This is not a problem: we will see how far we get.

Installation

You have to download and install *Eclipse for RCP and RAP Developers*. We have experience with Galileo and Helios. It looks like Indigo will work too.

Install IMP prerequisites from:

- IMP: http://download.eclipse.org/technology/imp/updates/0.2/

Install Rascal + IMP from:

- 5lang11: http://www.cwi.nl/~storm/5lang11/update

Some notes:

- 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 space will not work.

To start, open the Rascal perspective. Create a new project for the summer school. Download http://www.cwi.nl/~storm/5lang11/5lang11.zip and add the files to the project's src directory.

To open a Rascal console right-click on a Rascal editor or directory in the package outline, in the context-menu you'll find Launch Console.

Currently, after you have made a change in a Rascal module, you'll have to re-import the module in the console for the changes to take effect.

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

To create a grammar for state machines, create a new Rascal module, and add rules to recognizes the syntax invented by Martin Fowler (page 3 of the article cited above).

To get up and running add the following definitions for layout (white-space and comments) and identifiers:

```
syntax Id = lex [a-zA-Z][a-zA-Z0-9_]* - Reserved # [a-zA-Z0-9_];
syntax Reserved = "events" | "end" | "resetEvents" | "state" | "actions" ;
syntax LAYOUT = lex whitespace: [\t-\n\r\ ]  | lex Comment ;
layout LAYOUTLIST = LAYOUT* # [\t-\n\r\ ] # "/*" ;
syntax Comment = lex @category="Comment"  "/*" CommentChar* "*/" ;
syntax CommentChar = lex ![*] | lex Asterisk ;
syntax Asterisk = lex [*] # [/] ;
```

Quiz: what is the meaning of the # and - constructs? Why are they needed?

Don't forget to add a start syntax rule. This will instruct the parser what to recognize when a file is parsed in an editor. For instance, like this:

```
start syntax Controller = controller: Events ResetEvents? Commands? State+ states;
```

After you've created the syntax module. Create a module Plugin.rsc in the src directory of the project. Add the following lines to make the syntax available to the environment:

```
import <you syntax module>;
import util::IDE;
import ParseTree;

public void main() {
  registerLanguage("Controller", "ctl", <Your start symbol>(str input, loc org) {
     return parse(#<Your start symbol>, input, org);
  });
}
```

Open up a Rascal console, enter "import Plugin;", then "main();". If all is well, you can now open ".ctl" files and get syntax highlighting.

Tip: use the @Foldable attribute on a production alternative to get folding behaviour in the editor.

You can also parse from within the Rascal console:

```
import <Your syntax module>;
import ParseTree;
parse(#<Your start symbol>, "....")
```

The result will be a concrete syntax tree: a tree representing the structure of the input string. As you can see, it is very verbose: it includes *all* information about the source, including white-space and comments. Sometimes it can be tedious to work with such trees. For this reason, one often defines an *abstract* syntax tree which omits details that are irrelevant and leaves only the structure of the source. This is the next step.

Quiz: what is the meaning of # in the use 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 language. The Rascal standard library defines a function "implode" that turns a concrete syntax tree into an abstract syntax tree.

To make this work, define an ADT that corresponds 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) maps to the str data type.

You can now convert concrete syntax trees to ASTs as follows:

```
pt = ... parse tree of previous snippet ....
import <Your AST module>;
implode(#<Your AST module>::<Root ADT type>, pt);
```

Not that you'll have to qualify the root type of the ADT with the module name since the name will otherwise clash with the start symbol of your grammar.

Exercise: make a dedicated implode module which hides this complexity, so that you can just use implode(pt) without specifying the types.

From now on, we will work with AST values primarily.

Simple 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.

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:

```
alias TransRel = rel[str from, str token, str to]
alias ActionRel = rel[str state, str token]
```

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 token, and s’ the target state. The second relation captures which tokens should be output upon entering a certain state. Note that both relations use tokens and not the names of events or actions. This means that in your extraction you should take care of looking up event/command names to find the associated tokens.

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

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. You visualize a state machine by providing the relations of the previous assignment to the visualizer as follows. Import the module ShowStateMachine and then call the function:

```
void stateMachineVis(TransRel trans, ActionRel arel, str init)
```

supplying it with the two relations of the previous exercise and the initial state.

In the visualization there are buttons for each event. You can click on them to interactively simulate the execution of the state machine. To do this, the visualizer uses a step function based on the transition relation. This step function could be defined as follows:

```
public tuple[str state, list[str] output] step(str s1, str token,
               TransRel trans, ActionRel actions) {
   if (<s1, token, s2> <- trans) {
      return <s2, [ a | <s2, a> <- actions ]>;
   }
   return <s1, []>;
}
```

Exercise: write an interpreter for state machines based on this step function. This function should take a list of tokens and return a tuple of a final state and a list of output tokens.

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

The Message data type accepts source locations. They can be retrieved from AST nodes if you add an annotation declarations to your AST module for each ADT type:

```
anno loc <ADT type>@location;
```

Now you can obtain an AST node's source location using the aNode@location notation. The locations in the Message data type are used by the IDE to do error marking.

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.

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

Connecting the checker to the IDE

To hook up your checker to the IDE you have to add the following line to main in Plugin.rsc:

```
registerAnnotator("Controller", check);
```

The check function (which you'll have to provide), is expected to take a concrete parse tree and return an annotated parse tree. Concrete parse trees can be annotated with a set of Messages. In the check function you'll have to invoke your checker, get the collection of error message, and annotate the parse tree with it. You can annotate a tree using x[@a=v], where x is the tree, a is the annotation name and v is the annotated value.

Code-generation to Java

State machines have to be executed in code somehow. One approach is to generate (Java) code. In this assignment you are to write a function that generates Java 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><}>"

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.

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?

Connecting the code generator to the IDE

To be able to invoke the code generator from the state machine editor, add the following lines to main in Plugin.rsc.

```
contribs = {popup(menu("Controller,[action("Generate Java", generate)]))};
registerContributions("Controller", contribs);
```

You'll have to write a generate function that actually invokes your code generator and writes the result to file. Take a look in util::IDE (in the Rascal Eclipse standard library) to find out the expected signature for generate. Functions for input/output can be found in the standard library IO.

Parallel merge of two state machines

The desugaring reset events (cf. above) is an instance of a simple model-to-model transformation. In this assignment we will engage in a model-to-model 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 sate <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: write an interpreter that evaluates two state machines in parallel, without performing the parallel merge.

Exercise: use string templates to write an unparser for state machine ASTs so that you may inspect the result of the parallel merge as source code in an editor.

Bonus: Rename Refactoring

In the first assignment we mentioned that the use of ASTs is often more convenient for processing a language since it omits a lot of irrelevant detail that is present in the concrete syntax tree. However, for some applications this detail *is* important. Refactoring is one such application: after a refactoring, you don't want to have discarded all layout and comments! Hence, when implementing a refactoring we cannot use the AST, but have to use concrete parse trees.

Rascal provides “edit” contributions to context menus (cf. hooking up the code generator above). Such contributions accept closures that take a concrete parse tree, and source location indicating the current selection in the editor. They should return a string with the (possibly) rewritten source. The goal of this refactoring is to implement the “rename” refactoring for events, commands and states.

Some hints and guidelines:

- Use prompt from the util::Prompt module to ask for a new name. Use alert to signal that something is wrong.

- Use the treeAt function (from ParseTree) to find the source tree that corresponds to the selection. (Use pattern matching on the result to find out the type of the thing that was selected).
- Use the parse function (from ParseTree) to parse the new name (which is a string) into proper Identifiers.
- Think about the preconditions for the refactoring: when is it valid to rename something (assuming the state machine is well-formed)?
- Think about where a renaming should be applied. For instance, if an event is renamed, you should update the event declaration and any transition that fires on the event.
- Rename could be applied to events, states, and commands. Implement the refactoring for one kind of thing first. Then generalize.
- Use the visit statement to rewrite the concrete syntax tree. You may use concrete syntax patterns to match in cases. For instance you could use (Transition)‘<Id e> => <Id s>‘ to match a transition (depending on your grammar of course). Note that Rascal has to merge the grammar of the object language (state machines) and the grammar of Rascal to be able to parse such patterns. This may take some time.
- If you add labels to the elements of productions in your grammar, you can access sub-trees of (typed) parse trees as if they are fields. For instance, for additive expressions:

  syntax Exp = add: Exp lhs “+” Exp rhs

On a parse tree pt of type Exp, you can now evaluate pt.lhs and pt.rhs. Additionally you can test if a parse tree is of a particular alternative using: pt is add.

You may hook up the rename functionality to the IDE by extending the the pop-up menu actions as follows:

```
contribs = {popup([ ..., edit("Rename...", rename)])};
```