Design and Implementation of Domain Specific Languages

rascal-0.40.17

One of the goals of Rascal is to allow the design and implementation of Domain-Specific Languages.

In this usecase description we give a flavor of how you can use Rascal to:

1. define the syntax of a DSL;
2. create a simple semantic check;
3. how to compile the DSL to Java;
4. instantiate a VScode extension or Eclipse Plugin for the DSL.

The following example shows how to define a simple DSL for "state machines". It includes a parser, a check for unreachable states and a compiler to Java code.

Syntax definitions and parsing

The grammar of the DSL is defined using Rascal's grammar formalism which is fully integrated into the language.Shown below is the syntax definition of a simple state machine language, inspired by Martin Fowler's example language for gothic security.

- ❶ A state machine consists of a number of named state declarations
- ❷ each state contains transitions to other states (identified by name) when a certain event happens.

The grammar reuses identifier syntax and whitespace convention from the standard library, see Layout and Id. Each non-terminal defines a *type* and a *parser*. Parse trees are typed values like any other value in Rascal. Let's use a parser to parse this example state machine:

Functions that analyze parse trees

So now the DSL code has been transformed to a tree to type `Machine`. As a result, you can write functions that process such trees. An example function, would be a semantic check on state machines, such as finding all unreachable states:

To check for unreachable states, we first create a binary relation between states using a comprehension. This comprehension uses *concrete syntax* matching to find a state's transitions.

- The pattern between backticks is written in the object language, which in this case is the statemachine language defined in the grammar above.
- The variables `q1` and `ts` in between < and > are bound for each state that is found in the machine `m`.

- `"<q1>"` means the parse tree q1 is reduced to the literal string that it parsed (interpolated into an empty string)
- A similar pattern is used to find the target state `q2` is found in each transition in `ts`.
- The post-fix `+` then computes the transitive closure of the relation.

The relation `r` is based on the transitions in a state machine. This means that it does not include declared (final) states which have no outgoing transitions. We collect the names of all defined states in `qs` , again using a comprehension.

The initial state is (conventionally) defined to be the state that is declared first. An unreachable state is then defined as a state that is not in the right image of the initial state in the transitive closure of the transition relation. This is exactly what is described by the last comprehension! The notation `r[x]`, where `r` is a relation is short hand for `{ y | <x, y> <- r }`.

Let's take her for a spin:

Generating source code

There are various ways of compiling a DSL to target code in Rascal. The easiest and most direct way is using Rascal's string templates to generate code in a general purpose language. The following snippet shows the

generation of a Java while loop to execute a state machine.

String templates allow arbitrary Rascal values and control-flow constructs to be interpolated in string literals. Note how this code does not use concrete matching, but instead uses the labels defined in the grammar (i.e., `states`, `out`, `event`, and `to`).

Let's have a look at the output of `compile`:

And that's it! A complete DSL in 36 lines of code. Of course, the parser and the `unreachable` and `compile` functions can be connected to the IDE. This provides custom syntax highlighting, error-marking and automatic building in state machine editors.

Constructing an IDE

- For Eclipse Rascal offers plugin generation via the `util::IDE` module. You can register the language and the respective analysis, compilation and visualization functions with a single function call. The standard library module `util::IDEServices` allows for calling into features of the IDE (such as starting an editor or applying a refactoring).
- For VScode an interface with similar features and abstraction level is offered in `util::LanguageServer`, but geared towards the *Language Service Protocol*.