The Rascal Approach to Code in Prose, Computed Properties, and Language Extension

Solutions to the Language Workbench Challenge 2016

Pablo Inostroza
Centrum Wiskunde & Informatica, The Netherlands
pvaldera@cwi.nl

Tijs van der Storm
Centrum Wiskunde & Informatica, The Netherlands
University of Groningen, The Netherlands
storm@cwi.nl

Abstract

This document describes solutions in Rascal to three language workbench challenges, based on the questionnaire language QL: code in prose, computed properties, and language extension.

1. Introduction

Rascal is a language and environment for metaprogramming. One of the main application scenarios is as a textual language workbench for defining new programming languages or domain-specific languages [1, 7–9, 11, 12]. In this short paper, we present a brief description of Rascal, and then describe solutions to three language workbench challenges, based on the questionnaire language QL [3]: code in prose, computed properties, and (independent) language extension [4].

The code for all three challenges can be found online here: `https://github.com/cwi-swat/demoqles`.

2. Rascal

Rascal is a functional programming language designed for the metaprogramming domain. Being a functional language, it features immutable data structures, algebraic data types, comprehensions, pattern matching and higher-order functions. Regarding Rascal's specific metaprogramming capabilities, it features concrete syntax trees and source locations as native data types, a visit statement for structure-shy traversal, primitives for relational analysis (e.g. transitive closure, image, etc.), and sophisticated string interpolation for code generation.

To showcase metaprogramming in Rascal, we present a simple definition of a tiny DSL for state machines, inspired by Martin Fowler's *gothic security* example [5], which compiles the state machines into Java code[1].

First, we need to define the syntax of the State Machine language:

```
extend lang::std::Layout;

extend lang::std::Id;

start syntax Machine = machine: State+ states;
syntax State = @Foldable state: "state" Id name Trans* out;
syntax Trans = trans: Id event ":" Id to;
```

Besides extending the definitions of layout and identifiers from the standard library, we define concrete syntax types for machines, states and transitions using Rascal's built-in grammar formalism. These productions also defined types of values (concrete syntax trees).

To generate Java code from a state machine we use concrete syntax matching and string interpolation. This compile function is specified using the following three cases:

```
str compile(Machine m) =
  "while (true) {
   '  event = input.next();
   '  switch (current) {
   '    <for (q ← m.states) {>
   '      <compile(q)>
   '    <}>
   '  }
   '}";

str compile((State) `state <Id name> <Trans* ts>`) =
  "    case \"<name>\":
   '      <for (t ← ts) {>
   '      <compile(t)>
   '      <}>
   '      break;";

str compile((Trans) `<Id event> : <Id to>`) =
  "if (event.equals(\"<event>\"))
   '        current = \"<to>\";";
```

Note how concrete syntax pattern matching support pattern matching using the actual syntax of the defined language to destructure incoming syntax trees. This oversimplified example shows how Rascal's dedicated metaprogramming features aid in developing DSLs.

The Rascal IDE is implemented as an Eclipse plugin, and provides extension hooks into Eclipse to programmatically register IDE services for your own language. For example, in order to get syntax highlighting and syntactic error reporting for our State Machine language, we simply register the

[1] See `http://usethesource.io/dsl-in-36-lines-of-code`.

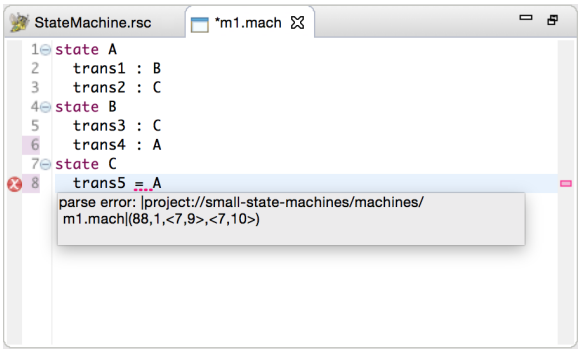

Figure 1. Rascal-based State Machines IDE

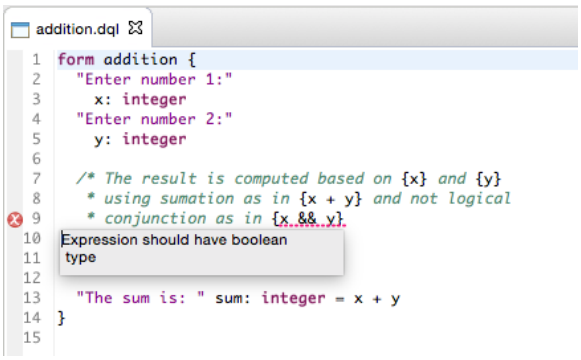

Figure 2. Code in prose in the QL IDE

language to the IDE, by indicating how a given source string is to be parsed into a Machine. Notice that the parse function, provided by the standard library, is parameterized in the syntactic types of a grammar (in this case Machine).

```
registerLanguage("STM", "stm", start[Machine](str s, loc l) {
  return parse(#start[Machine], s, l);
});
```

IDE services based on analysis and transformation are realized by registering contributions. For instance, one may want to trigger the compilation to Java whenever a state machine file is saved. This can be achieved using the "builder" contribution. In this case, the builder simply calls the previously defined compile function and saves the result to a file:

```
registerContributions("STM", {
  builder(set[Message] (start[Machine] s) {
    writeFile(pt@\loc[extension = "java"].top, compile(s));
    return {};
  })
});
```

Figure 1 shows the generated state machine IDE in action, featuring syntax highlighting and syntaxc error reporting.

The following sections are dedicated to explain how have we addressed three of the challenges proposed in [4].

3. Code in prose

We showcase how to embed code in prose by adding expressions within comments in QL programs. Figure 2 shows that variables x and y are referenced in the comment (enclosed in curly braces), as well as the expression x + y and x && y. Since the latter is not type-correct, the Rascal IDE indicates the corresponding error. Moreover, since the name analyzer analyzes the expressions within the comment, they are hyperlinked to navigate to their definition.

Assumptions Our implementation of the "code in prose" challenge is based on two key techniques.

First, Rascal's parsing engine is based on scannerless parsing [14], which means that there is no distinction between tokenization and parsing. Thus, it is possible to define comments using ordinary context-free syntax rules, allowing the inclusion of "ordinary" syntactic symbols, such as expressions.

```
layout MyLayout = Comment | Whitespace

lexical Whitespace = [\u0009–\u000D \u0020 \u0085] ;

syntax Comment = CStart CommentChar* CEnd;

syntax CStart = @category="Comment" "/*";
syntax CEnd = @category="Comment" "*/";

syntax CommentChar
  = @category="Comment" ![*{}\ \n\t]
  | @category="Comment" [*] !>> [/]
  | "{" Expr expr "}";
```

Figure 3. Simplified grammar for embedding expressions in comments.

Second, parsing a program using a grammar defined in Rascal produces concrete syntax trees, which include all textual information from the source code, including whitepsace, keywords, and comments. This information is thus available throughout all kinds of analyses and transformations.

Implementation Figure 3 shows a slightly simplified definition of layout of QL. The **layout** keyword captures the syntax that is allowed in between syntactic elements. In this case it consists of comments or whitespace. Comments in the QL grammar are defined as the non-terminal Comment, using the keyword **syntax**, which introduces a context-free symbol. This means that layout is allowed in between elements. The actual interpolation is defined in the last production of the CommentChar non-terminal.

The Rascal IDE framework derives syntax highlighting styles from grammar annotations. For instance, the annotation `@category="Comment"` is used to highlight comments as shown in Figure 3. Note however, that the expression interpolation is not annotated to obtain default hightlighting of embedded expressions.

To obtain error marking and jump-to-definition hyperlinks the name analyzer and the type checker traverse comments and simply analyze and check the embedded expressions.

Variants Our solution only shows "singe-hole" interpolation of expression inside comments. It is also possible to imagine structured interpolation, where parts of the interpolated expression can be interleaved with free form prose. A well-known example of this is conditional and loop interpolation in template languages. In terms of the grammar such interpolations can be easily expressed; however, the semantic analysis of such fragments would require some more effort, since it requires custom patterns not available in the "pure" host language.

Usability Impact on usability is minor. What will be considered as prose will behave just like ordinary code. The main limitation is that users will have to type delimiters – such as the curly braces in our embedding example – to mark the transition out of the comment into an expression, and vice versa.

Impact In order to embed expressions in prose, the only artifact that needs to be changed is the grammar. If the comments had already been defined as `syntax` non-terminals, then the non-terminal representing the code embedding simply has to be added as a valid component of a comment, including changes to reject (unescaped) delimiters from the main comment text. All components that have to analyze the embeddings will have to be modified to explicitly traverse into the comments.

Composability The layout nodes (which typically contain comments) in Rascal's concrete syntax trees are normally ignored. For instance, concrete syntax patterns are matched modulo layout. This means that embedding code in comments has no effect on existing code, except if one likes to process the expressions embedded in comments themselves, for instance to analyze them. All other operations, however, remain oblivious to the embedded fragments.

Limitations The technique requires delimiters to disambiguate code from prose.

Uses and Examples Rascal itself features string templates, which contains arbitrary expression interpolation, as well as conditional and loop interpolation. These string templates are implemented in a similar style.

Effort The effort of implementing code in comments for QL consisted of refactoring the comment grammar along the lines of Figure 3, and adapting name analysis and type-checking. Together this took roughly 30 minutes.

4. Computed Properties

QL features ordinary questions and computed questions. In this challenge we enrich QL with the possibility for the user to provide inputs to questions, right from within a QL program. The IDE will then evaluate the questionnaire and insert the result of computed questions in the source text.

Figure 4 shows a simple questionnaire for computing the average of two numbers. The first two questions are

```
form Average {
  "First:" x: integer [10]
  "Second:" y: integer [2]
  "Average:" avg: integer = (x + y) / 2 [6]
}
```

Figure 4. Computed properties in QL: the result value of the last, computed question is inserted by the IDE, based on the input values provided by the user for x and y.

annotated with two input literal values (10 and 2), within square brackets. The evaluator will consider these values to be the actual, "run time" values of those questions. The computed property, in this case, is the literal annotation of the last question (6). This one is not provided by the user, but inserted by the QL IDE, after evaluating the questionnaire.

Assumptions Concrete syntax trees. Although not essential, it makes patching the text editor contents much more reliable; and AST based approach would require more advanced origin tracking or "diff" technology (see, e.g., [2]).

Implementation The computed properties (values in this case) are represented as ordinary syntax, so the grammar of QL needs to be changed. The following excerpt from the Rascal QL grammar shows how input values (for ordinary questions), and output values (the actual computed properties) for computed questions are represented:

```
syntax Question
  = question: Label Var ":" Type Value?
  | computed: Label Var ":" Type "=" Expr Value?;

syntax Value = "[" Const "]";
```

The user can now "enter" values in the questionnaire by providing a value in the source text of a QL program. After every such change the questionnaire will be evaluated and analyzed for changes in the value of computed questions. If there are changes they are reflected in the Value positions of computed questions in the editor.

- **Evaluation** proceeds in three steps: 1) create the initial environement, mapping all questions to default values; 2) updating the environment with user supplied values; 3) constructing the final environment by evaluating computed questions in a fixed point computation.
- **Patch generation** takes the form and the final environment, and for each computed question compares the value in the form (if any) to the corresponding value in the environment. If it is different, it adds a textual update operation to the resulting patch.

The evaluation component is standard, and completely modular from this particular challenge; it can be generally reused as an interpreter of QL expressions.

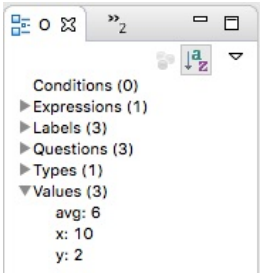

Figure 5. Showing computed properties as part of the outline view

Patch generation creates a patch structure which is defined by the following Rascal type: **lrel**[**loc**, **str**]. This type represents lists of tuples with a source location (**loc**) and a string. The source location captures exact positions inside some resource (e.g., a source file); it includes offset, length, line and column information). The concrete syntax trees used in Rascal are annotated by the same kind of source locations. When the patch generation algorithm determines it needs to "patch" a computed value in the source tree, it uses these locations to determine the exact substring of the source file that needs to be changed. A patch tuple can thus be read as "change this source location to that string".

After the patch has been generated, the IDE will use it to modify the contents of the editor accordingly. To achieve this, the code realizing the above is invoked from the "liveUpdater" editor contribution. Slightly simplified, this is achieved with the following Rascal code:

```
registerContributions("Demoqles", {
  liveUpdater(lrel[loc,str] (start[Form] pt) {
    return patch(f, evalForm(f));
  }),
  ...
});
```

Whenever there's a change to the editor contents, the closure passed to the "liveUpdater" value is invoked, returning the (possibly empty) patch to the editor framework.

Variants Our current implementation is based on modifying the source text directly. Different approaches could use other existing IDE affordances to show the computed properties. For instance, it would be quite easy to show it as "info" markers, hover documentation, or in the outline. In fact, our implementation shows the actual values of the questions in a separate section of the outline (Figure 5).

Each of these approaches is easier to implement than the syntax-based approach we have demonstrated, since they do not require syntax extension and patch generation. Nevertheless, such variants have different usability characteristics and reduced flexibility.

For instance, the outline is a separate view, creating distance between the actual values and the code. Hover documentation only appears on-demand. And info markers instantly show up in the editor as squiglies and gutter markers, but to see the contents of the message one still has to hover over the marker.

Regarding flexibility, the syntax-based approach can show any kind of information, structured or not, as long as it is possible to design a suitable syntax. Hover documentation, information markers or outlines are much more limited in that they typically support only string-based values, or untyped tree structures.

In the described challenge we have only shown how live values are shown as part of the text, but not the visibility of questions as follows from the conditional logic in questionnaires. We have, however, built an extension which changes the syntax of conditional blocks to trigger a different (subdued) highlighting. This shows how the source code itself communicates which questions are visible or not given a particular assignment of values to questions.

Usability As to usability, the experience is exactly as before, except that some parts of the source code are automatically computed and provided by the user. There is no change to how the editor behaves: copy-paste still works, everything can be edited, saved etc.

However, whenever a computed property is modified by the user, the change will be immediately overridden. Note that this is semantically perfectly fine, however, it may be experienced as an awkward glitch in the experience. A possible extension to avoid this, would be to integrate live updating with support for (non-)editable regions, which we have experimented with in earlier research [6].

Usability furthermore depends on how "nice" the computed properties are in the source code. For instance, to avoid requiring the user to constantly reformat the computed properties, the patch generation algorithm may need to carefully take into account indentation and whitespace. Given that our implementation is based on concrete syntax trees in the first place, all information is available to achieve the right layout, but it may be cumbersome for the developer.

Impact Existing constructs now exist in different variants (e.g., questions with input annotations, and without), so components that perform case-based analysis on the syntax tree need to be extended in order skip or deal with these extended variants. In our case, for instance, name resolution simply ignores the additional `Value`, but still needs to analyze questions with values attached. Alternatively, the type checker does not simply ignore the input annoations, but actually type checks it.

The final change is to hook up the patch generation to the live updating extension point of the IDE. The other components (expression evaluation, patch generation) are modular additions.

Note that the impact on existing language components is mainly due to having to define explicit syntax for the input and output values. In more elaborate languages (e.g., programming languages), it would be possible to use existing syntax for this [13]. For instance, in a language with syntax

function calls, an invocation of "special" functions could conventionally be used to specify inputs and outputs. In such cases, modifying existing components could be avoided.

Composability Apart from the required syntax extension, no other components or language features are affected by this feature. Evaluating the form, and producing the patch are simply two additional interpretations of the source progam. Existing components like the compiler will just process ordinary source code.

Note that for this challenge we did not opt for a fully modular implementation. However, as we hope to show in Section 5, Rascal's module system is powerful enough to allow both syntax and interpretations (name resolution, type checking etc.) to be modularly extended to support the new value annotations.

Limitations Our solution is fully text-based, so the limits on the presentation of computed questions are defined by what can be represented as text and parsed using Rascal's grammar formalism.

Since Rascal's grammar formalism is backed by generalized parsing, language extensions can be modularized. A limitation is that as a consequence, there's also the risk of creating an ambiguous grammar. The language developer thus needs to be careful that computed properties do not introduce ambiguities.

Uses and Examples Rascal itself features experimental scrapbook pages: source editors that can be used as an interactive canvas for experimenting with Rascal. Whenever a statement is entered, the output of its evaluation is represented as part of the source code. In a sense, this represents a read-eval-print-loop (REPL), fully within source code. The computed properties are the outputs (values, standard output, or errors) of the entered commands.

CellDown is an experimental spreadsheet language designed by the second author, which shows views of a spreadsheet where all formulas are evaluated as part of the CellDown source text itself. This time the computed properties are complete spreadsheets. As of now, this is still unpublished work.

The implementation shown here represents an example of "Live literals" [13], where the source code of a program is used as input and output of dynamic information. A similar mechanism was used to implement live tests, spreadsheets, and probes [10] in Javascript.

Effort The current implementation was realized in about 3 hours, consisting of modifying the syntax to include input and output syntax, modifying the outliner, implementing the expression evaluation and patch generation. Together, the latter two components take up 134 SLOC.

5. Language extension

In this challenge we extend QL with a conditional unless construct, without changing existing code.

Assumptions Rascal's module system supports extensible syntax and extensible operations. We assume that the operations are defined using pattern-based dispatch [1]. This means that each syntactic case distinction made in an operation correspond to a single Rascal function definition, dispatching on that syntactic construct using pattern matching. Such functions can be extended by simply adding additional definitions dispatching on the new constructs.

Furthermore, we don't desugar unless to **if**(not(..)). While this would provide for a straightforward, modular implementation, it would present a problem for operations which need access to concrete representation of the source code (e.g., to generate good errors, formatting, the patch generation of Section 4, etc.).

Implementation The first component is the extension of the grammar:

```
module lang::demoqles::unless::Unless
extend lang::demoqles::ql::QL;

syntax Question = "unless" "(" Expr expr ")" Question;
```

This module adds the alternative for unless to the Question non-terminal.

As an example of extending an operation, this is the code for extending the type checker:

```
module lang::demoqles::unless::Check
import lang::demoqles::unless::Unless;
extend lang::demoqles::ql::Check;

set[Message]
  tc((Question)`unless (<Expr c>) <Question q>`, Info i)
   = tcCond(c, i) + tc(q, i);
```

The extended Check module defines the functions tc and tcCond. The former is extended here to add type checking of the unless construct. The definition simply calls the tcCond function to type check the condition and recursively calls tc to type check the body. The compiler (which produces HTML and Javascript code) is extended in a similar fashion, as is the patch generation code for controlling the question visibility.

With these extensions, the entry points for the parser, type checker etc. are the new modules. In fact we now have modularly defined a *new* language, which includes the old one. This means that the IDE services need to be wired again, using the extended language components. This is similar to writing a new "main" method, and does not compromise the modularity of the language extension.

Variants Extensions like unless are quite simple, affecting only the grammar, type checker, patch generation and compiler. More interesting language additions involve extending name resolution, outlining and others. As long as these components are defined using case-based analysis, the operations can be extended.

If, on the other hand, a new language feature also requires extensions to internal data structures (symbol tables etc.),

a modular implementation would be more complicated. A similar complication arises when language components depend on each other, for instance when name analysis and type checking are interleaved, this requires mutually open recursive definitions, since both components will have to be extended, and both components need to "see" each others extensions as well as their own.

Usability There is no effect on usability of the editor. The only restriction now is that the extended language needs to have a different name and file extension.

Impact No code had to be changed; the unless implementation is realized by strictly adding new code, with only minor duplication in the "main" wiring of the IDE services.

Composability Our implementation of unless involves a modular extension to the computed properties challenge described in Section 4. Furthermore, the **extend** feature of Rascal's module system is not limited to linear extension. Modularly combining unless with another independent extension is possible.

Limitations Currently, there can be no dependencies between two operations through **import**, if both operations need to be extended. For instance, if the type checker imports and invokes name resolution, and both are extended with new cases, then the extended type checker will only see the non-extended name resolution. This is an area of ongoing work.

Uses and Examples The pattern to modularly extend QL with unless has been used in the implementation of Oberon-0, a small, imperative programming language for use in compiler construction courses [1][2].

Effort The unless implementation was developed in approximately 1 hour. The implementation consists of 22 SLOC, with an additional 66 SLOC for top-level wiring.

References

[1] B. Basten, J. van den Bos, M. Hills, P. Klint, A. Lankamp, B. Lisser, A. van der Ploeg, T. van der Storm, and J. Vinju. Modular language implementation in Rascal—experience report. *Science of Computer Programming*, 114:7 – 19, 2015.

[2] M. de Jonge and E. Visser. An algorithm for layout preservation in refactoring transformations. In *SLE'11*, pages 40–59. Springer, 2011.

[3] S. Erdweg, T. Van Der Storm, M. Völter, M. Boersma, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh, et al. The state of the art in language workbenches. In *SLE'13*, pages 197–217. Springer, 2013.

[4] S. Erdweg, T. van der Storm, M. Völter, L. Tratt, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh, G. Konat, P. J. Molina, M. Palatnik, R. Pohjonen, E. Schindler, K. Schindler, R. Solmi, V. Vergu, E. Visser, K. van der Vlist, G. Wachsmuth, and J. van der Woning. Evaluating and comparing language workbenches: Existing results and benchmarks for the future. *Computer Languages, Systems & Structures*, 44, Part A:24–47, 2015.

[5] M. Fowler. *Domain Specific Languages*. Addison-Wesley Professional, 1st edition, 2010. ISBN 0321712943, 9780321712943.

[6] P. Inostroza, T. Van Der Storm, and S. Erdweg. Tracing program transformations with string origins. In *ICMT'14*, pages 154–169. Springer, 2014.

[7] P. Klint and R. Van Rozen. Micro-machinations: a DSL for game economies. In *SLE'13*, pages 36–55. Springer, 2013.

[8] P. Klint, T. van der Storm, and J. J. Vinju. RASCAL: A domain specific language for source code analysis and manipulation. In *SCAM*, pages 168–177. IEEE, 2009.

[9] P. Klint, T. van der Storm, and J. Vinju. EASY meta-programming with Rascal. In *GTTSE III*, volume 6491 of *LNCS*, pages 222–289. Springer, 2011.

[10] S. McDirmid. Usable live programming. In *Onward! 2013*, pages 53–62. ACM, 2013.

[11] J. van den Bos and T. van der Storm. Bringing domain-specific languages to digital forensics. In *ICSE SEIP*, pages 671–680. ACM, 2011.

[12] T. van der Storm. The Rascal Language Workbench. CWI Technical Report SEN-1111, CWI, 2011.

[13] T. van der Storm and F. Hermans. Live literals. In *LIVE'16*, 2016. `http://www.cwi.nl/~storm/livelit/livelit.html`.

[14] E. Visser. Scannerless generalized-LR parsing. technical report. Technical Report P9707, Programming Research Group, University of Amsterdam, 1997.

[2] `https://github.com/cwi-swat/oberon0/tree/ldta-only`