Bacatá: A Language Parametric Notebook Generator

Mauricio Verano Merino
Eindhoven University of Technology
Eindhoven, The Netherlands
m.verano.merino@tue.nl

Jurgen Vinju
CWI
Amsterdam, The Netherlands
jurgen.vinju@cwi.nl
Eindhoven University of Technology
Eindhoven, The Netherlands

Tijs van der Storm
CWI
Amsterdam, The Netherlands
storm@cwi.nl
University of Groningen
Groningen, The Netherlands

Abstract

Interactive notebooks allow people to communicate and collaborate through a single rich document that might include live code, multimedia, computed results, and documentation, which is persisted as a whole for reproducibility. Notebooks are currently being used extensively in domains such as data science, data journalism, and machine learning. Constructing a notebook interface for a new language, however, requires a lot of effort. In this paper, we present Bacatá, a language parametric notebook generator for domain-specific languages (DSL) based on the Jupyter framework [15]. Bacatá is designed so that language engineers may reuse existing language components (such as parsers, code generators, interpreters etc.) as much as possible. We present the design of Bacatá and how DSL notebooks can be generated with minimum effort in the context of the Rascal meta programming system and language workbench. We demonstrate Bacatá's utility by generating notebook interfaces for three languages, Halide* (a DSL for image processing), SweeterJS (an extended version of Javascript), and QL (a DSL for questionnaires). Our results show that notebooks generated by Bacatá often only require a few lines of code to wire existing components together.

Keywords Computational narratives, interactive computing, language workbenches, domain-specific languages, literate programming

1 Introduction

Interactive notebooks have received much attention in the recent years due to the benefits they provide regarding immediate feedback, reproducibility, and collaborative features. Notebooks capture a *computational narrative* interleaving code, computed results, interactive visualizations, and documentation, in a single persisted document. Notebooks have become very popular in fields such as mathematics, data science, data journalism, and machine learning.

The Jupyter notebook framework [15] is a popular platform for writing and sharing computational narratives. This platform comes with built-in support for Python, but it provides an API for extending the framework with other languages, called "kernels". Language kernels capture language specific aspects, such as how to highlight syntax elements, how to call the interpreter or compiler, and how to visualize computed results.

Developing a language kernel from scratch requires a lot of effort, and requires communicating with Jupyter's low-level wire protocol. Nevertheless, interactive notebooks would provide a valuable addition to the toolbox of generic language services offered by language workbenches [6]. This would open up the interactive notebook metaphor for DSLs developed using these language workbenches.

In this paper we present Bacatá, a language parametric notebook generator, based on the Jupyter platform. Bacatá hides the low-level complexity of Jupyter's wire protocol, providing generic hooks for registering language services. Bacatá has been integrated in the Rascal language workbench [13], which allows extensive reuse of language components defined with Rascal. As a result, obtaining a notebook interface for a DSL becomes a matter of writing a few lines of code. In addition, Bacatá supports fully interactive computed results through Rascal's web UI framework (Salix). DSLs that exploit this library in their execution can thus be run from within a Bacatá notebook, with virtually no additional effort.

The contributions of this paper can be summarized as follows:

- We motivate notebooks from the perspective of DSL use and DSL engineering, and provide a feature-based analysis of interactive notebooks (Section 2).
- We present Bacatá-Core, a generic language protocol in Java to simplify the development of Jupyter language kernels (Section 3).
- We present Bacatá-Rascal as a light-weight bridge between Bacatá-Core and Rascal, and show how this API can be used to generate notebooks for DSLs developed in Rascal (Section 4).
- Bacatá's utility is demonstrated by generating notebooks for three languages: Halide [20] (a DSL for image processing), SweeterJS (an extended version of JavaScript), and QL [6] (a DSL for defining questionnaires) (Section 5).

We conclude the paper with a discussion of related work and future directions of research (Section 6 & 7).

SLE'18, November 04–09, 2018, Boston, MA, USA
2018.

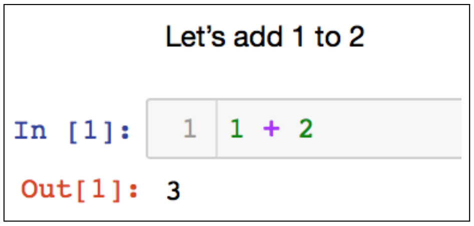


Figure 1. A basic notebook.

2 Background

2.1 Anatomy of a Notebook

Notebooks enable users to teach, learn, and share knowledge by telling a story. Storytelling is a pedagogical strategy and a robust communication and collaboration tool [5]. Notebooks are useful for computational story telling because they interleave documentation, input, and output in a single linear document. In its most simple form, a notebook consists of a sequence of cells that can be categorized in three types: prose cells for documentation, input cells containing code, and output cells displaying computed results.

An example is shown in Figure 1. The first row consists of prose text explaining what is going to happen. The second row displays an input cell where the user has entered the expression "`1 + 2`" in some programming language. Finally, the last row shows the output of evaluating the expression.

Notebooks are interactive: readers can tweak input parameters, change code snippets, and observe different ways of representing the output. For instance, changing the expression in the input cell will trigger the recomputation of the current output cell. More advanced styles of notebooks feature interactive visualizations of computed results as well, which support interactive exploration of (large) data sets.

Notebooks are persisted as a single document, which facilitates sharing computational narratives. Furthermore, since all documentation, input, and output is part of the notebook, results can be reliably reproduced.

2.2 Notebooks for DSLs

Most existing interactive notebooks (e.g., for Python, R, Julia), are based on full-fledged programming languages. Domain-Specific Languages (DSLs), however, are often small languages tailored to particular problem domains. They are designed as a way of communication between domain experts and software engineers. This raises the question of why it is important to consider developing notebooks for DSLs. Below we analyze the reasons why DSL users and DSL engineers may benefit from interactive notebooks.

Non-programmer use. Unlike general-purpose programming languages, DSLs are often used by domain experts who are not necessarily proficient in software development or computer science. Interactive notebooks provide a more friendly interface for interacting with computation than full-fledged IDEs or basic text editors. In addition, the fact that notebooks run from ordinary web browsers avoids installation hassle. In summary, notebooks make for a less intimidating software development engineering.

Experimentation and simulation. Interactive notebooks deviate from the traditional software development setting where the goal is to create production quality software, towards a setting where exploration and experimentation take center stage. In the context of DSLs, this allows domain experts to experiment with the language, enjoying immediate feedback and reproducibility, without the pressure of software engineering concerns. As soon as the design and requirements are stabilized, notebooks can provide input to production-level code generators that create the actual software. As such, notebooks reinforce the division of labor between domain engineers and application engineers promoted by Domain-Specific Software Engineering (DSE) [2].

Notebooks for DSL education. DSLs are typically small languages, designed for a specific audience, developed by smaller teams than general purpose programming languages like Java or C#. As a result, the use of DSLs incurs costs regarding documentation and training. Notebooks can function as live tutorials, providing interactive walk-throughs for a DSL. Notebooks may thus complement standard forms of documentation (e.g., user guides, reference manuals, API documentation, etc.), to allow domain experts to familiarize themselves with a new DSL.

Language engineering benefits. The engineering trade-offs in the construction of DSLs are different from general-purpose programming languages. DSLs are often developed in-house, by smaller teams, and requiring a faster design iteration cycle. Notebooks can provide a valuable tool in the language engineer's toolbox for testing and debugging a language implementation. Especially since various language engineering aspects can be exposed as part of the notebook. For instance, as we will show in Section 5, notebooks can display outputs of language implementation components such as generated code, static analysis results, test results, etc.

2.3 Notebook Features

To analyze the generic and language specific aspects of a notebook, we have performed a feature-oriented domain analysis to capture the features to be supported by notebooks. Figure 2 shows the resulting feature diagram [11]. The root of Figure 2 represents the characteristics to be supported by notebooks. Some of the features in the diagram may appear either as mandatory or optional A description of each feature in Figure 2 is presented below.

Highlighting Syntax highlighting differentiates characters and words according to their role in the programming

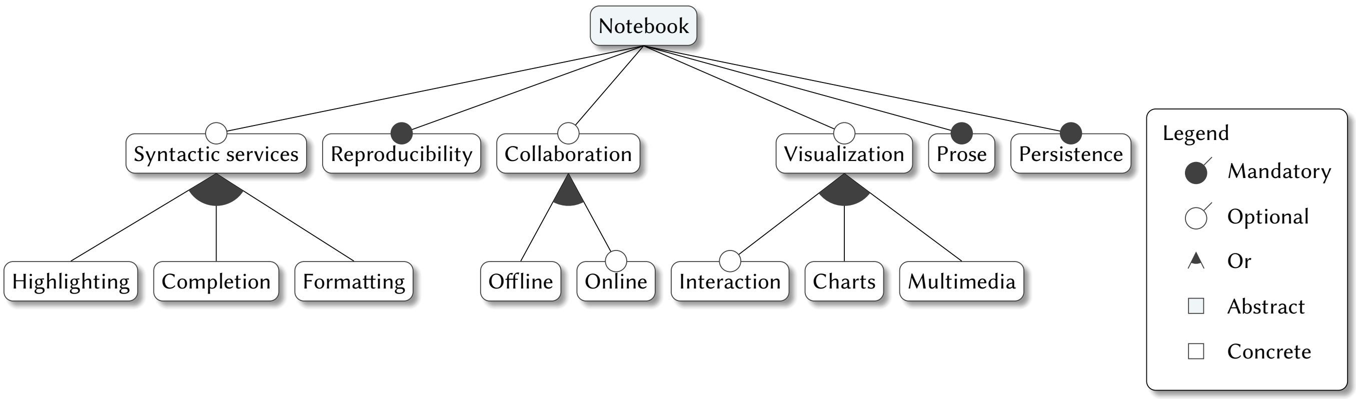


Figure 2. Feature-oriented domain analysis of notebooks.

language syntax. This is similar to standard syntax highlighting in IDEs and language workbenches, and improves the readability of the code by letting users visualize differentiate language elements such as keywords, data types, identifiers, among others.

Completion Completion provides the users with suggestions to complete an incomplete fragment of source code. This can be syntactic – template completion – or semantic, based on the scoping rules and variables/functions in a language. Syntactic completion is especially useful for discoverability of language features when learning a new language. Semantic completion helps avoiding errors in referring to defined entities.

Formatting is the visual form in which code and prose is presented to the end-user. Thus, code and prose becomes more readable and maintainable.

Reproducibility Notebooks are often used in a scientific context. As a result, reproducibility is important for peer review and verification. Notebooks can contain both the story and development of a scientific result, and have is the ability to reliably reproduce previous interactive computations using the same data, code, prose, to obtain identical results.

Collaboration Notebook can be easily shared to have multiple users working on the same notebook. Each one may be focused on different aspects or sections of it. Again, this feature is supported by the single document metaphor offered by notebooks.

Visualization Notebooks do not necessarily only support textual output, but often feature rich visualization capabilities to present information in various ways such as graphs, charts, images, animations, or even full-blown interactive Graphical User Interfaces (GUIs).

Prose Next to code fragments, notebooks allow users to interleave live code and documentation using prose cells. Therefore, users will be able to describe their experiments in a linear storytelling way, using different languages for marking up documents such as LaTeX, *Markdown*, and *HTML*.

Persistence All information in a notebook is persisted in a single file. This includes all the code, input data, documentation, and computed results. Additionally, notebook results can also be stored on external files as a side effect of the cell execution, for some language kernels.

Summary. Looking at the feature model we can observe that some features are language-specific and some are independent of the actual language. The following features are in the first category: highlighting, completion, formatting, and visualization. The other features – reproducibility, collaboration, prose, and persistence – are orthogonal to the language-specific features and are handled generically by notebook frameworks such as Jupyter.

Apart from visualization, perhaps, the language specific features are already part of the standard toolset of language workbenches [6]. In the following section we describe Bacatá, a language parametric framework for generating interactive notebooks based on the Jupyter framework, designed to reuse existing language workbench features for obtaining the language specific notebook features.

3 Bacatá

Bacatá is a language parametric interface between the Jupyter platform and the Rascal language workbench. This interface generates Jupyter language kernels that reuse language components such as grammars, parsers, and Read-Eval-Print Loops (REPLs). In this section, we explain Bacatá's language service interface and how it reuses language components. Then we describe Bacatá's general architecture.

3.1 Architecture

Figure 3 depicts a general overview of Bacatá's architecture, which highlights its most essential components. Two primary actors interact with Bacatá, language engineers and end-users. Language engineers use Bacatá to generate a

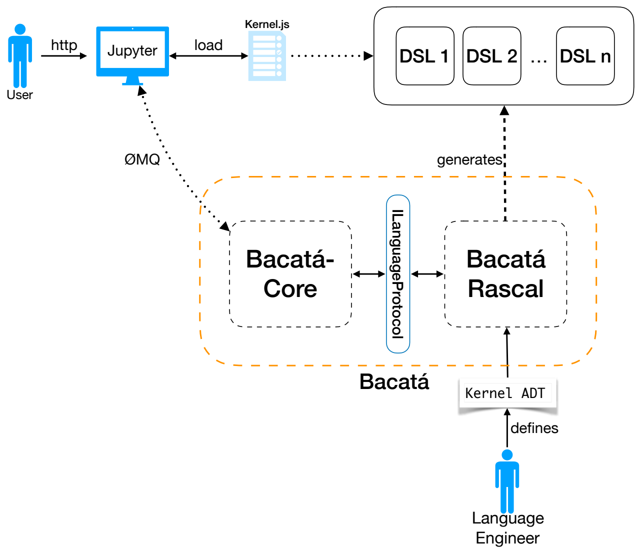


Figure 3. General overview of Bacatá's architecture.

Jupyter language kernel. Whereas end-users utilize a language kernel, previously generated by a language engineer, to interact with the language through a notebook front-end.

Bacatá consists of two main components, Bacatá-Coreand Bacatá-Rascal. On the one hand, *Bacatá-Core* abstracts away the communication layer between Jupyter and the language. It provides a generic language protocol interface (similar to Microsoft's Language Server Protocol [18]), that could be implemented for language workbenches other than Rascal. This component is responsible for the interaction between the executable code written in a notebook and its execution.

On the other hand, Bacatá-Rascalimplements the interface offered Bacatá-Core, and provides the means for languages developed using Rascal to be connected to Bacatá-Core. To use those services, Bacatá-Rascaltakes as input an Algebraic Data Type (ADT) called `Kernel`. A `Kernel` object is the entry-point for generating and re-using language-specific artifacts such as CodeMirror [8] modes, language interpreters, completion functions, and interactive visualizations. After a language engineer generates a language kernel using Bacatá, this language automatically becomes part of the supported languages of the Jupyter environment.

From the end-user perspective, Bacatá-Rascaland Bacatá-Coreare invisible, since they simply choose their desired language kernel from the Jupyter notebook interface. After selecting the language kernel, Jupyter automatically instantiates the language REPL through Bacatá, which allows the user to execute code.

3.2 Bacatá-Core

Jupyter offers a protocol called the *wire protocol* [10], which is a communication protocol implemented using ZeroMQ sockets [1]. This protocol describes a set of sockets and messages that enable the interaction between third-party languages and the Jupyter platform. Similarly, it describes the structure of the messages and how to interchange those messages among the different sockets used by Jupyter. To extend Jupyter's default set of languages, language engineers need to implement a *language kernel*. A language kernel is a program that runs user code. To create a language kernel from scratch, language engineers have to communicate with the low-level wire protocol.

```
data Kernel
  = kernel(str language, loc project,
      str replFunction, loc logo = |tmp:///|);
```

Listing 1. Kernel ADT.

```
data REPL
  = repl(Result(str) handler,
         Completion(str) completor);

alias Completion
  = tuple[int pos, list[str] suggestions];

data Result
  = text(str result, list[Message] messages);
```

Listing 2. REPL ADT.

Bacatá-Core offers the `ILanguageProtocol` interface that enables the communication between Jupyter and a language in a generic way. The primary purpose of this layer is to abstract the implementation complexity of the wire protocol and its related socket management. Therefore, the language developer can focus on the language engineering layer. For DSLs developed within Rascal, we have implemented an this interface in a language parametric way. In other words, it pretends to be a particular language kernel, but delegating all language specific service requests to a language implementation in Rascal.

4 Bacatá-Rascal

4.1 Introduction

As explained before, to support new languages by Jupyter, developers have to implement a language kernel. Bacatá is a Jupyter language kernel generator for DSLs written within the Rascal LWB.

To use Bacatá's kernel generator, a language engineer needs to define a function that produces a `REPL` ADT, which will be used as the language's interactive interpreter. The `REPL` ADT is defined as shown in Listing 2.

1. The language engineer calls the Bacatá function `bacata` which accepts one argument, a value of type `Kernel`.

The `Kernel` type (shown in Listing 1), defines the configuration parameters for Bacatá-Core to obtain language specific information (e.g., name and location of the logo of the language) and find relevant resources, such as the fully qualified name of the REPL implementation to be used.

2. The generated kernel assumes that there is a `replFunction` which returns a `REPL` value. The `REPL` data type is shown in Listing 2. It encapsulates two functions, the `handler` for interpreting code, and a `completor` for code completion. The respective result types of each function are also shown in Listing 2.
3. Optionally, language engineers can generate CodeMirror syntax-highlighting modes. This can be achieved by providing a value of the data type `Mode` (Listing 6), which can optionally be automatically derived from the language's grammar.

The function `bacata` takes a `Kernel` object to generate a JSON file called *kernel.json* (cf. Listing 5). This file contains different data such as Jupyter's connection details (e.g., ZMQ socket types), language REPL execution instructions, and language-specific information (e.g., name and logo). When an end-user request to generate a notebook for a specific language, all this data is being forwarded to Bacatá. Then, after generating the JSON file, Bacatá automatically registers the language as part of the Jupyter supported languages.

4.2 A Full Example: Calc

Now that we have seen the basic components of Bacatá, let us explore a complete example of generating a notebook for a simple calculator language (Calc). The definition of Calc is shown in Listing 3. It defines the syntax of the language using Rascal's built-in grammar formalism. The language consists of commands (`Cmd`) and expressions (`Exp`). Commands consist of assignments and expression evaluation. Expression forms are variables, numbers, multiplication, and addition. Commands are executed using the `exec` function, which returns a number and a (possibly updated) environment. Expressions simply evaluate to numbers.

Given the language definition of Listing 3, we can now define a function that creates a `REPL`, as shown in Listing 4. The function `myRepl` contains two functions, `myHandler` and `myCompletor`. The handler function receives the user input, tries to parse it as a `Cmd`, and then executes it. If parsing was successful, a `text Result` (Listing 4) is returned with the computed result. Otherwise, the handler returns an empty result with an error message corresponding to the parse error. The function `myCompletor` iterates over the variables defined in the environment `env`, and returns the variables that partially match with the `prefix`, together with the index `pos` where the match in the prefix starts. Finally, both the handler and the completor are wrapped as a `REPL` value and returned.

```
module Calc
extend lang::std::Id;
extend lang::std::Layout;

syntax Cmd = Id "=" Exp | Exp;

syntax Exp
  = Id | Num | left Exp "*" Exp > left Exp "+" Exp;

lexical Num = [\-]?[0-9]+;

alias Env = map[str, int];

tuple[int, Env] exec(Cmd cmd, Env env) { ... }

int eval(Exp exp, Env env) { ... }
```

Listing 3. Definition of Calc

```
module Repl
import Calc;

REPL myRepl() {
  Env env = ();

  Result myHandler(str line) {
    try {
      Cmd cmd = parse(#Cmd, line);
      <n, env> = exec(cmd, env);
      return text("<n>", []);
    }
    catch ParseError(loc l):
      return text("", [message("Parse error", l)]);
  }

  Completion myCompletor(str prefix)
    = <pos, [ x | x <- env, startsWith(p, x) ]>
    when /<p:[a-zA-Z]*$/ := prefix,
         pos := size(prefix) - size(p);

  return repl(myHandler, myCompletor);
}
```

Listing 4. A REPL implementation for Calc

Note that the code of both Listing 3 and Listing 4 is independent of Bacatá and Jupyter. The syntax definition and evaluator function can be reused in different contexts as well. Similarly, `REPL` can also be used for an ordinary command line interface, for instance, as an interactive console in the IDE. The same code is used by Bacatá to generate a Jupyter notebook.

The following interactive session at the Rascal console shows how to generate a Jupyter kernel with Bacatá, using the `REPL` function in Listing 4:

```
{"argv": [
    "java", "-jar",
    "/Mauricio/bacata/bacata-dsl.jar",
    "{connection_file}",
    "home:///projects/Calc",
    "Repl::myRepl",
    "Calc"
   ],
   "display_name": "Calc",
   "language": "Calc"}
```

Listing 5. Generated Jupyter kernel for Calc

```
data Mode
  = mode(str name, list[State] states);

data State = state(str name, list[Rule] rules);

data Rule
  = rule(str regex, list[str] tokens,
         str next = "", bool indent = false,
         bool dedent = false);
```

Listing 6. Syntax Mode ADT

```
> k = kernel("Calc", |project://Calc|, "Repl::myRepl");
>> ...
> nb = bacata(k);
>> ...
> nb.serve();
The notebook is running at: |http://localhost:8888|
```

We first create a `Kernel` value, consisting of the language name, the project location, and the qualified name of the `REPL` function. The `bacata` function generates the Jupyter *kernel.json* file (shown in Listing 5) and returns a notebook value, which can then be started within the same session. Alternatively, the notebook server can also be started from the commandline outside of Rascal.

4.3 Syntax Highlighting

Jupyter's input cells highlighting is based on the CodeMirror editor[1], which supports easily customizable syntax highlighting through the use of *modes*. Modes are similar to so-called "Textmate grammars"[2], which are used by editors such as Textmate, VS Code, SublimeText, and many others.

The `Mode` data type shown in Listing 6 models such modes. A mode has a name and contains a number of state definitions. Each state then defines a number of rules that are applicable in that state. A rule defines a regular expression to match a particular substring and assigns a list of token types to it that will determine its visual appearance. After a rule has matched, it may transit to another state via the `next` property. The optional booleans `indent` and `dedent` control auto indentation in block constructs.

To support syntax highlighting in Bacatá-generated notebooks, the `bacata` function supports an optional additional argument for the mode:

```
Notebook bacata(Kernel k, Mode mode=mode("", [])) {...}
```

A simple mode for the Calc language could look as follows:

```
mode("Calc", [state("init", [
  rule("[0-9]+", ["number"]),
  rule("[a-zA-Z][a-zA-Z0-9_]*", ["variable"])])])
```

This mode defines a single state with two rules for numbers and variables.

Language engineers can define such modes manually. However, Bacatá also features a function to generate simple modes for keyword highlighting from a Rascal grammar using reflection.

4.4 Interactive Visualizations

Jupyter notebooks run in the browser, so this allows output cells to contain almost arbitrary interactive visualizations, beyond the simple text output that we have seen in the Calc example. Bacatá supports fully interactive, stateful graphical user interfaces in output cells through integration with Rascal's web UI framework Salix[3]. Salix supports all the standard HTML and SVG elements, and features integration with graph rendering libraries[4], and chart frameworks[5].

A Salix application is encapsulated as a value of type `App[&T]` where the type parameter `&T` indicates the type of the application data model. Under the hood, an `App` encapsulates a view to draw UIs using HTML and SVG elements, and an update function to update the model when a user event is triggered, respectively. Bacatá makes use of such Salix applications by allowing Salix `Apps` as output of the REPL. This is achieved by extending the `Result` data type of Listing 2:

```
data Result
  = ...
  | app(App[&T] app, list[Message] messages);
```

This kind of result can be used to produce fully functional stateful output cells, leveraging all UI features of Salix.

To illustrate the flexibility of `app`, we can extend the Calc language with a very simplified expression debugger to visualize the effect of variables on expression evaluation. The first step is to extend the language with another command to trigger the visualization:

[1] https://codemirror.net
[2] https://manual.macromates.com/en/language_grammars
[3] https://github.com/cwi-swat/salix
[4] https://github.com/dagrejs
[5] https://developers.google.com/chart/

```
data Msg = var(str x, str val);

App[Env] expApp(Exp e, Env env) {
  Env init() = env;

  void view(Env env) {
    div(() {
      for (str x ← env) {
        text("<x>: <env[x]>");
        input(\type("range"), \value(env[x]),
              onInput(partial(var, x)));
      }
      text("<e>: <eval(e, env)>");
    });
  }

  Env update(var(x, v), Env env) = env + (x:toInt(v));

  return makeApp(init, view, update);
}
```

Listing 7. Expression debugger defined using Salix.

```
syntax Cmd
  = ...
  | "show" Exp;
```

So, for instance, if the user types **show** x + y, a debugger of the expression x + y will appear in the output cell.

The debugger itself is defined as the Salix application shown in Listing 7. The environment serves as the application model. The Msg data type encapsulates events supported by the application; in this case there's only one, capturing change of variable's value in the environment.

The function expApp then defines the actual application. It consists of three nested functions. The first produces the initial model, in this case the environment env passed into expApp. The view function takes an environment and draws the UI. The debug view will consist of rows of sliders for each variable in the environment, producing var messages when the user modifies the slider position. Finally, the expression itself is shown as text together with the value it evaluates to. Finally, the update function updates the model, in this case represented by the environment.

The last required modification consists of having the REPL return an expApp when the user enters the show-command. This is achieved by adding the following statements, just after parsing the command:

```
Cmd cmd = ...
if ((Cmd)`show <Exp e>` := cmd) {
  return app(expApp(e, env), []);
}
```

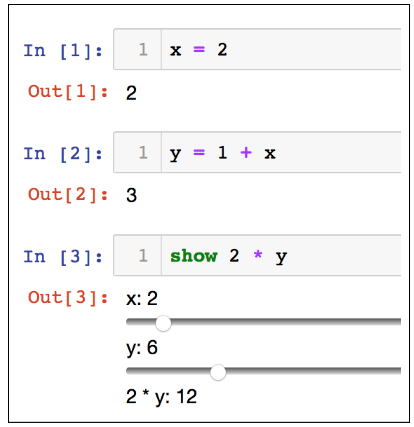


Figure 4. Interactive debugging of a Calc expression.

The **if**-condition uses Rascal's concrete syntax pattern matching to check if cmd is a show-command, binding e to the argument expression. If the match succeeds, the app result containing the App produced by expApp is returned.

The resulting debugging interface is illustrated in Figure 4. The user has typed in two assignments to variables x and y, and then invokes the show-command to inspect the effect of the current variable bindings on the expression 2 * y. The result is two slider widgets for variable x and y, together with current evaluation of 2 * y. When changing the slider for y the new result will be live updated on the last line.

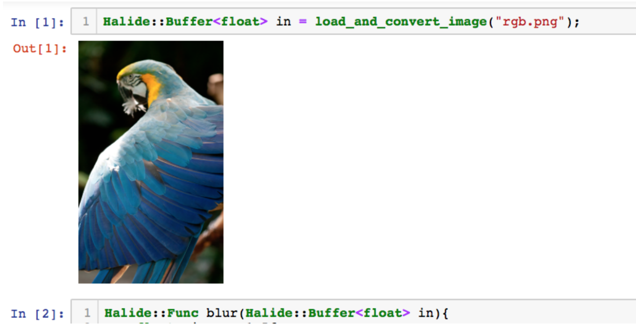
5 Case Studies

We have implemented a notebook interface using Bacatá for DSLs developed using the Rascal LWB. The DSL interface was used to generate notebooks for three different languages Halide*, SweeterJS, and QL.

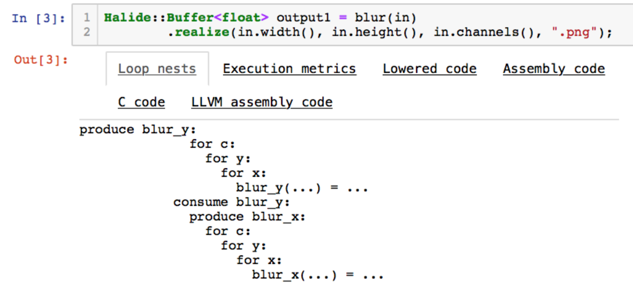
5.1 Halide*

Halide [20] is a language for image processing and computational photography. To generate a Halide notebook, we have implemented Halide*, a subset of Halide, implemented in Rascal. Halide* was explicitly designed to be used within a notebook environment, due to the order of steps required for the construction and execution of image processing pipelines.

This DSL is used to generate, compile, and execute native Halide source code; the Halide compiler does the compilation and execution steps. In Halide* we have introduced some syntactic sugar such as function wrappers to be able to differentiate between main functions, image pipeline definitions, compilation strategies (e.g., ahead of time or just in time (JIT) compilation), and execution. The syntactic sugar was added to make the execution of Halide code more amenable to the notebook style of working. The execution of the Halide code through the notebook client is done by calling the Halide



(a) Loading an image and defining a function `blur` (snipped).



(b) Calling `blur` and inspecting generated artifacts.

Figure 5. Halide notebook.

compiler to execute user's code. Bacatá intercepts those results, parses them into HTML, and then displays them within the output cells of the Halide* notebook.

A prototypical session using the Halide* notebook is shown in Figure 5. The case study highlights multi-media outputs and inspection of generated compiler artifacts. On the left (Figure 5a) the user loads an image, which is directly shown as the output result. Then a function `blur` is defined (snipped), which does not produce output but is now available for use. Then, on the right (Figure 5b), the `blur` function is invoked on the input image `in`. The result shows a tabbed interface to inspect loop nests, execution metrics, lowered code, assembly code, C code, and LLVM assembly code.

5.2 SweeterJS

To illustrate the benefits of notebooks from the language engineering perspective, we have generated a notebook interface for SweeterJS, a variant of Javascript for teaching language extension through source-to-source transformation (desugaring) using Rascal[6]. Using the notebook students can enter snippets of extended Javascript, and see both the computed result and the desugared source code.

An example is shown in Figure 6 where the user has entered some Javascript code with an SQL-like query expression (line 5). Evaluating the cell produces the actual output of running the desugared Javascript code, but also shows the desugared code itself. In this case, the query expression is transformed to a JSLINQ query constructor.

5.3 Questionnaire Language (QL)

The last used language is QL, which is a DSL for building interactive questionnaires. QL has been used to benchmark and evaluate language workbenches [6] and is interesting from the perspective of notebooks since QL programs define interactive GUI applications.

A questionnaire consists of a form which may contain one or more questions. There are three different types of

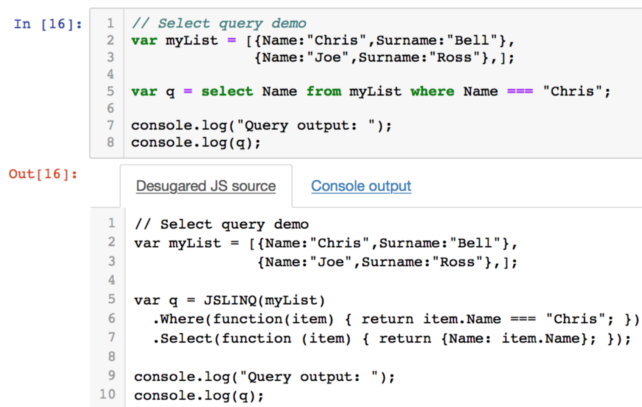


Figure 6. SweeterJS notebook showing desugared output.

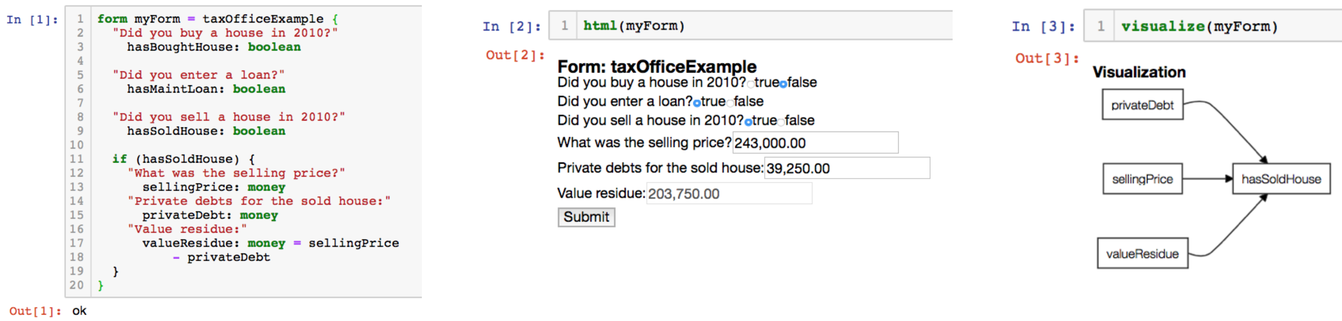
questions, namely labeled, conditional, and computed questions. Questionnaires can be executed as interactive HTML forms, which we implemented using the Salix library. Additionally, the QL notebook supports visualizing the control dependencies between questions, which is a valuable tool for questionnaire designers to understand the conditional logic of a questionnaire.

Figure 7 shows a sample interaction with the QL notebook using the example of a simple tax filing questionnaire. The user first defines a questionnaire `myForm` using the `form`-command (Figure 7a). Then, in Figure 7b, the form is rendered using the `html` command. Note that the output is a fully working questionnaire, as if it were deployed, so this allows easy and interactive testing of questionnaires. Alternatively, to understand the conditional logic of a form, the user can visualize the control dependencies using the `visualize` command (Figure 7c).

5.4 Effort

To assess the flexibility in creating Jupyter notebooks using Bacatá, we compare the number of Source Lines of Code (SLOC) that are independent of Bacatá to the number of

[6] https://github.com/cwi-swat/hack-your-javascript



(a) QL form definition. **(b)** QL form execution. **(c)** Question control dependencies.

Figure 7. QL notebook.

Language	Reused SLOC	Notebook SLOC
Calc	37	50
Halide*	51	647
SweeterJS	579	162
QL	771	120

Table 1. Reused vs new code in SLOC.

SLOC required to define the notebook itself. These results are shown in Table 1.

The Calc language is included as a simple baseline, and consists of the code discussed in Section 4.2. The reused code consists of the syntax definition and the `exec` and `eval` functions. The notebook code includes the definition of the REPL and the expression debugger Salix application.

The Halide* implementation differs from the others in that the code generators are designed specifically to be used in the notebook setting. As a result, the only reusable code is the syntax definition. This explains why the notebook code is larger.

In both SweeterJS and QL the ratio between reused and new code is much higher. In the case of SweeterJS, the reusable code includes the syntax definition of Javascript, language extensions for state machines, queries, and a variant of HAML[7], and the transformations to desugar the extensions to vanilla Javascript.

The interpreter for QL had already been defined using Salix, so could be reused directly. The same holds for the syntax definition, name resolver, and type checker. The new code includes the code for the REPL, and the control-dependency visualization.

As can be observed from Table 1, creating a notebook using Bacatá requires limited effort. The main component to be written is the function defining the REPL and the code completor, which basically consists of wiring existing components together.

[7] http://haml.info/

6 Related Work

Bacatá can be positioned in a long line of research in program environment generation [3, 6, 9, 12, 21, 23, 25]. Currently, this work is is centered around the concept of language workbenches, a term popularized by Fowler [7]. In his essay, he explains a brief history of the language-oriented programming, their pros and cons, and how IDE tooling has become essential for the viability of language oriented programming, and learning and using DSLs.

Language workbenches provide language parametric tools, meta languages, and techniques to lower the cost of DSL engineering. Bacatá aims to do the same for notebooks. Specifically, interactive notebooks provide a different user interface for code and documentation. Orthogonal to, but not in conflict with more traditional IDE or editor styles.

Concerning interactive computing, Cook [4] and Nagar [19] have highlighted the importance of this paradigm of software development. Cook [4], shows the consequences of adopting this paradigm and how it affects the way we write code based on immediate responses. While Nagar [19] shows a Python way of working using interactive computing, and how it has reduced the learning curve of a programming language if the user can experiment with commands and expressions.

Notebooks integrate the use of narrative in software development, literate programming [16, 22], interactive computing, and collaboration. Turner et al. [24] found notebooks useful as a way of supporting cooperative work and sharing information with non-technical staff. This is aligned with the perspective of using notebooks for DSLs that have a non-programmer audience. However, they found difficult to differentiate between formal an informal information. Similarly, Malony et al. [17] performed computational experiments using a notebook environment, called the Virtual Notebook Environment (ViNE) [**?**].

7 Conclusions & Future Work

Interactive notebooks provide a user interface for interacting with computational narratives, integrating code with documentation and live, interactive feedback. Unlike traditional IDEs and editors, notebooks focus on interactive exploration and computational story telling.

Constructing interactive notebooks for new languages requires a lot of effort, especially in the context of DSLs, where the engineering trade-offs and design cycle is different from general purpose language. In this paper, we have presented Bacatá, a language paramteric notebook generator based on the Jupyter framework. Given existing language components, such as parsers, interpreters, type checkers etc., Bacatá reduces the effort of obtaining an interactive notebook interface to writing a few lines of code wiring language components together.

We described the core architecture of Bacatá, and presented how the interface is exposed within the Rascal language workbench. Next to the usual notebook features (executing code, code completion, highlighting), we have shown how Bacatá supports fully interactive output cells using Rascal's web-based GUI framework Salix. The Rascal binding to Bacatá has been used to define notebook interfaces for three languages, Halide*, SweeterJS, and QL, exercising multiple aspects of the framework. Comparing the required number of new lines of code versus the number of lines of code that could be reused shows that Bacatá-generated notebook interfaces require little effort.

A main direction for future work is consolidating the `ILanguageProtocol` interface of Bacatá with Microsoft's Language Server Protocol [18]. This would allow DSL engineerings to implement a single IDE interface once and for all, which could serve both traditional IDEs, as well as interactive Jupyter notebooks.

References

[1] Faruk Akgul. 2013. *ZeroMQ*. Packt Publishing.

[2] Barrett R. Bryant, Jeff Gray, and Marjan Mernik. 2010. Domain-specific Software Engineering. In *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research (FoSER '10)*. ACM, New York, NY, USA, 65–68. https://doi.org/10.1145/1882362.1882376

[3] Philippe Charles, Robert M Fuhrer, Stanley M Sutton Jr, Evelyn Duesterwald, and Jurgen Vinju. 2009. Accelerating the creation of customized, language-Specific IDEs in Eclipse. In *ACM Sigplan Notices*, Vol. 44. ACM, 191–206.

[4] Joshua Cook. 2017. *Interactive Programming*. Apress, Berkeley, CA, 49–70. https://doi.org/10.1007/978-1-4842-3012-1_3

[5] Raffaele Di Fuccio, Michela Ponticorvo, Fabrizio Ferrara, and Orazio Miglino. 2016. Digital and Multisensory Storytelling: Narration with Smell, Taste and Touch. In *Adaptive and Adaptable Learning*, Katrien Verbert, Mike Sharples, and Tomaž Klobučar (Eds.). Springer International Publishing, Cham, 509–512.

[6] Sebastian Erdweg, Tijs van der Storm, Markus Volter, Laurence Tratt, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad Vergu, Eelco Visser, Kevin van der Vlist, Guido Wachsmuth, and Jimi van der Woning. 2015. Evaluating and comparing language workbenches: Existing results and benchmarks for the future. *Computer Languages, Systems & Structures* 44 (2015), 24 – 47. https://doi.org/10.1016/j.cl.2015.08.007 Special issue on the 6th and 7th International Conference on Software Language Engineering (SLE 2013 and SLE 2014).

[7] Martin Fowler. 2015. Language Workbenches: The Killer-App for Domain Specific Languages? (2015). Retrieved June 18, 2018 from https://www.martinfowler.com/articles/languageWorkbench.html

[8] Marijn Haverbeke. 2007–2018. CodeMirror. (2007–2018). http://codemirror.net/

[9] Jan Heering and Paul Klint. 2000. Semantics of Programming Languages: A Tool-oriented Approach. *SIGPLAN Not.* 35, 3 (March 2000), 39–48. https://doi.org/10.1145/351159.351173

[10] Jupyter. 2015. The wire protocol. (2015). Retrieved July 24, 2017 from http://jupyter-client.readthedocs.io/en/latest/messaging.html#the-wire-protocol

[11] Kyo Kang, Sholom Cohen, James Hess, William Novak, and A. Peterson. 1990. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Technical Report CMU/SEI-90-TR-021. Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA. http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=11231

[12] P. Klint. 1993. A Meta-environment for Generating Programming Environments. *ACM Trans. Softw. Eng. Methodol.* 2, 2 (April 1993), 176–201. https://doi.org/10.1145/151257.151260

[13] Paul Klint, Tijs van der Storm, and Jurgen Vinju. 2009. RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation. In *Proceedings of the 2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM '09)*. IEEE Computer Society, Washington, DC, USA, 168–177. https://doi.org/10.1109/SCAM.2009.28

[14] Clemens Nylandsted Klokmose and Pär-Ola Zander. 2010. Rethinking Laboratory Notebooks. In *Proceedings of COOP 2010*, Myriam Lewkowicz, Parina Hassanaly, Volker Wulf, and Markus Rohde (Eds.). Springer London, London, 119–139.

[15] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica Hamrick, Jason Grout, Sylvain Corlay, Paul Ivanov, Damián Avila, Safia Abdalla, and Carol Willing. 2016. Jupyter Notebooks – a publishing format for reproducible computational workflows. In *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, F. Loizides and B. Schmidt (Eds.). IOS Press, 87 – 90.

[16] Donald E. Knuth. 1984. Literate Programming. *Comput. J.* 27, 2 (May 1984), 97–111. https://doi.org/10.1093/comjnl/27.2.97

[17] Allen D. Malony, Jenifer L. Skidmore, and Matthew J. Sottile. 1999. Computational experiments using distributed tools in a web-based electronic notebook environment. In *High-Performance Computing and Networking*, Peter Sloot, Marian Bubak, Alfons Hoekstra, and Bob Hertzberger (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 381–390.

[18] Microsoft. 2018. Language Server Protocol. (2018). https://microsoft.github.io/language-server-protocol

[19] Sandeep Nagar. 2018. *IPython*. Apress, Berkeley, CA, 31–45. https://doi.org/10.1007/978-1-4842-3204-0_3

[20] Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. 2012. Decoupling Algorithms from Schedules for Easy Optimization of Image Processing Pipelines. *ACM Trans. Graph.* 31, 4 (July 2012), 32:1–32:12.

[21] Thomas Reps and Tim Teitelbaum. 1984. The synthesizer generator. *ACM SIGSOFT Software Engineering Notes* 9, 3 (1984), 42–48.

[22] Johannes Sametinger. 1997. *Literate Programming*. Springer Berlin Heidelberg, Berlin, Heidelberg, 211–216. https://doi.org/10.1007/978-3-662-03345-6_18

[23] Emma Söderberg and Görel Hedin. 2011. Building semantic editors using JastAdd: tool demonstration. In *Proceedings of the Eleventh Workshop on Language Descriptions, Tools and Applications*. ACM, 11.

[24] Phil Turner and Susan Turner. 1997. *Supporting Cooperative Working Using Shared Notebooks*. Springer Netherlands, Dordrecht, 281–295. https://doi.org/10.1007/978-94-015-7372-6_19

[25] M.G.J. van den Brand, A. van Deursen, J. Heering, H.A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P.A. Olivier, J. Scheerder, J.J. Vinju, E. Visser, and J. Visser. 2001. The ASF+SDF Meta-Environment: A Component-Based Language Development Environment. *Electronic Notes in Theoretical Computer Science* 44, 2 (2001), 3 – 8. https://doi.org/10.1016/S1571-0661(04)80917-4 LDTA'01, First Workshop on Language Descriptions, Tools and Applications (a Satellite Event of ETAPS 2001).