A Language-Parametric Approach to Exploratory Programming Environments

L. Thomas van Binsbergen
ltvanbinsbergen@acm.org
University of Amsterdam
Amsterdam, The Netherlands

Damian Frölich
dfrolich@acm.org
University of Amsterdam
Amsterdam, The Netherlands

Mauricio Verano Merino
m.verano.merino@vu.nl
Vrije Universiteit Amsterdam
Amsterdam, The Netherlands

Joey Lai
joeylai96@hotmail.com
University of Amsterdam
Amsterdam, The Netherlands

Pierre Jeanjean
pierre.jeanjean@inria.fr
Univ. Rennes, IRISA, Inria
Rennes, France

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

Benoit Combemale
benoit.combemale@irisa.fr
Univ. Rennes, IRISA, Inria
Rennes, France

Olivier Barais
olivier.barais@irisa.fr
Univ. Rennes, IRISA, Inria
Rennes, France

Abstract

Exploratory programming is a software development style in which code is a medium for prototyping ideas and solutions, and in which even the end-goal can evolve over time. Exploratory programming is valuable in various contexts such as programming education, data science, and end-user programming. However, there is a lack of appropriate tooling and language design principles to support exploratory programming. This paper presents a host language- and object language-independent protocol for exploratory programming akin to the Language Server Protocol. The protocol serves as a basis to develop novel (or extend existing) programming environments for exploratory programming such as computational notebooks and command-line REPLs. An architecture is presented on top of which prototype environments can be developed with relative ease, because existing (language) components can be reused. Our prototypes demonstrate that the proposed protocol is sufficiently expressive to support exploratory programming scenarios as encountered in literature within the software engineering, human-computer interaction and data science domains.

SLE '22, December 06–07, 2022, Auckland, New Zealand
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9919-7/22/12…$15.00
https://doi.org/10.1145/3567512.3567527

CCS Concepts: • **Software and its engineering → Development frameworks and environments**; *Interpreters.*

Keywords: Exploratory programming, protocol, IDEs, REPLs, notebooks, interpreters

ACM Reference Format:
L. Thomas van Binsbergen, Damian Frölich, Mauricio Verano Merino, Joey Lai, Pierre Jeanjean, Tijs van der Storm, Benoit Combemale, and Olivier Barais. 2022. A Language-Parametric Approach to Exploratory Programming Environments. In *Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering (SLE '22), December 06–07, 2022, Auckland, New Zealand.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3567512.3567527

1 Introduction

In traditional software development processes, a predefined set of requirements specifies what features the software must support under which circumstances. Exploratory programming[1], however, is an open-ended activity with no upfront specification. Exploratory programming is a style in which code is used as a medium for prototyping, and in which the goal and solution are to be discovered together through experimentation [3, 47, 56]. An essential characteristic of exploratory programming is experimentation within a design space, trying out different design alternatives by extending or tweaking programs. Programming environments can support this programming style by allowing programmers to create, edit, and evaluate (partial) programs.

In conventional IDEs, experimentation is often limited by the edit-compile-run cycle, which does not offer the desired experience in terms of feedback and responsiveness.

[1]The term 'exploratory programming' was coined by Beau Shiel in 1986 [52] and is sometimes also referred to as 'opportunistic programming'.

Command-line REPLs (Read-Eval-Print Loop), such as the Python interpreter and JShell REPL for Java, support incremental programming, in which a program is developed piecewise by entering and executing code fragments rather than full-fledged programs [60]. In REPLs, the effects of code fragments are immediately reported to the user, and, to varying extents, the current program state can be inspected, queried or accessed in various ways. As such, REPLs provide a better interface for experimentation than conventional IDEs. Computational notebooks, such as the MatLab environment [18] and Jupyter notebooks [26], go beyond REPLs by supporting re-executing, modifying, and/or copying previously executed code fragments stored in code cells, interleaved with documentation. Through documentation cells, literate programming [27] is combined with incremental programming, making it easier to communicate ideas with collaborators. However, computational notebooks are still limited from the perspective of exploratory programming (e.g., lack of feedback, reusability, and information about notebook's execution state) [10, 16, 20, 22, 23, 46, 50].

Motivating Example. Figure 1 shows a prototype of an exploratory programming environment for QL, a DSL for specifying questionnaires. The left-hand side shows a code editor and a running application view, representing the currently active version of the code and the current run-time state of the rendered questionnaire, respectively. The right-hand side of the figure shows a tree-structured trace of all interaction with either the code or the running questionnaire, in the form of code cells containing commands.

The figure displays the end-result of the following steps:

- The programmer defines the first question "What is your age?". Source edits are reflected in the REPL history as semantic deltas [61] (cell `3d58`).
- She then tries it out by entering the value 42 in the run-time view, resulting in the command `age = 42`.
- To experiment with computed questions, she forks off a branch by pressing the right-arrowed button. In the sub-tree (headed by cell `7ead`), she enters a question that computes $2 \times age$.
- Satisfied with the effect, the programmer moves back to the main branch using the lightning button, jumping back to `2a08`, the last command of the main line.
- The height question is then added, again resulting in a semantic delta in the history.
- Next, the programmer wants to experiment with conditional questions. Another temporary branch is created, with another computed question, which is conditional on *height > 200*.
- Finally, she returns to the main branch, which ended at the entry of the `height` question. This is the state that is shown on the left of the figure.

The QL prototype demonstrates various interesting features related to exploratory programming. Firstly, multiple branches of code execution can be explored, and previous states can be revisited, without losing work. In this way, multiple variants of a program can be developed simultaneously such that their effects can be compared. Secondly, the prototype shows that certain actions in the interface (e.g., entering an age) result in executed code that can (therefore) also be undone. Our goal is to reduce the effort required for engineering programming environments with features like these for (new) software languages, DSLs in particular.

Contributions. This work reports on the next step in a research line aimed at designing and implementing features that simplify exploratory programming in (general-purpose and domain-specific) programming environments. Previous work has demonstrated how software languages can be (re-)designed to enable exploratory programming with 'execution graphs' encoding execution histories as a central datastructure [11, 33, 60]. This paper contributes by presenting a *protocol* for interacting with execution graphs alongside an *architecture* designed for prototyping exploratory programming features. We evaluate the protocol by discussing the extent to which it supports exploratory programming scenarios encountered in the literature. The design, efficient implementation, and evaluation of features for exploratory programming are future work. The main contributions are:

1. We extend the foundations provided by earlier work (Section 2) to support divergent programs and to improve the handling of program output (Section 3).
2. We present the Exploratory Programming Protocol (EPP, Section 4) and an architecture (Section 7) with a potential for reducing the effort of engineering prototype environments for exploratory programming.
3. We discuss the expressivity of the protocol (Section 6) in relation to various exploratory programming scenarios encountered in the literature (Section 5).

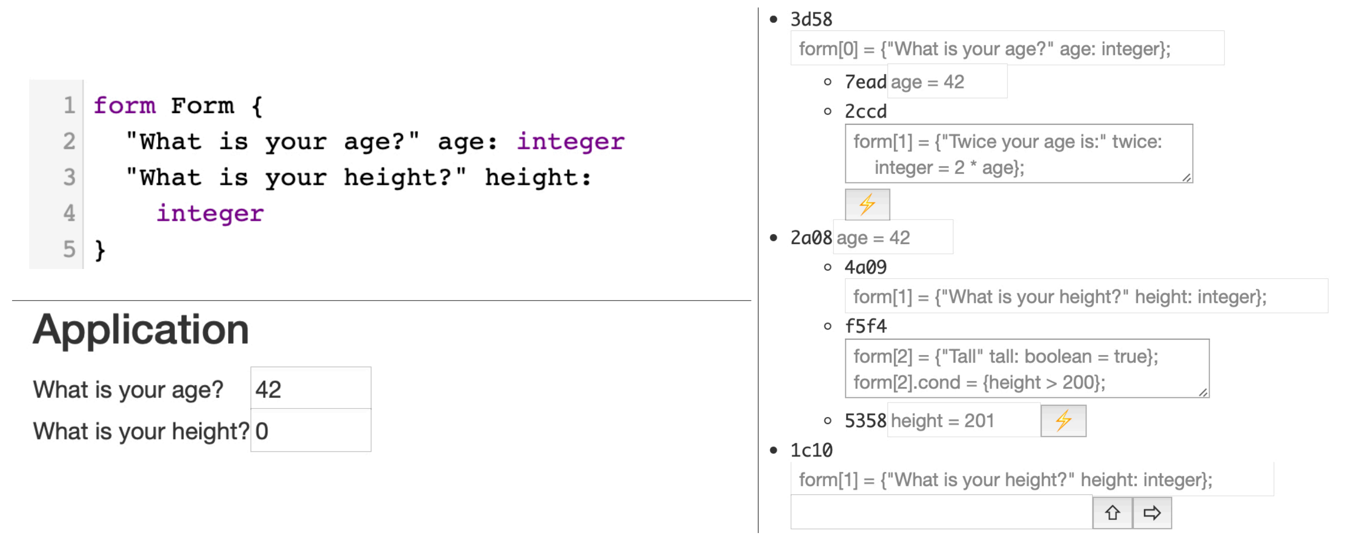
We discuss strengths and limitations in Section 8, related work in Section 9 before concluding in Section 10.

2 Background

The foundations of the protocol presented in this paper are provided by the principled approach to REPLs of [60]. In this section we describe the main concepts using a basic calculator language as an example object language defined in Rascal [25]. In the next section we extend the approach based on observations made in [11]. The (abstract) syntax of the calculator language is as follows:

```
                                    data Expr
data Cmd                              = add(Expr lhs, Expr rhs)
  = expr(Expr e)                      | mul(Expr lhs, Expr rhs)
  | assign(str x, Expr e);            | var(str x)
                                      | lit(int n)
```

In the approach, a language is defined by its abstract syntax, the syntax of configurations, an initial configuration,



```
form Form {
  "What is your age?" age: integer
  "What is your height?" height:
    integer
}
```

Application

What is your age? 42

What is your height? 0

- 3d58
  form[0] = {"What is your age?" age: integer};
  - 7ead age = 42
  - 2ccd
    form[1] = {"Twice your age is:" twice: integer = 2 * age};
- 2a08 age = 42
  - 4a09
    form[1] = {"What is your height?" height: integer};
  - f5f4
    form[2] = {"Tall" tall: boolean = true};
    form[2].cond = {height > 200};
  - 5358 height = 201
- 1c10
  form[1] = {"What is your height?" height: integer};

Figure 1. Exploring QL with branching time and fine-grained versioning. Top left: the source input; bottom left: the running application; right: the exploration trace.

```
Config exec(expr(Expr e), Config c)
  = <eval(e, c.store), c.store>;
Config exec(assign(str x, Expr e), Config c)
  = <v, c.store + (x: v)>
  when int v := eval(e, c.store);
int eval(var(str x), Store s) = s[x];
int eval(lit(int n), Store s) = n;
int eval(add(Expr lhs, Expr rhs), Store s)
  = eval(lhs, s) + eval(rhs, s);
int eval(mul(Expr lhs, Expr rhs), Store s)
  = eval(lhs, s) * eval(rhs, s);
```

Figure 2. Evaluation functions for the calculator language.

and a definitional interpreter. Configurations contain all results produced by executing a program and all contextual information needed to execute a program. For example, in the calculator language, configurations contain a result value and a store mapping variables (strings) to (integer) values.

```
alias Store = map[str var, int val];
alias Config = tuple[int result, Store store];
Config initial = <0, {}>;
```

Evaluation functions are given in Figure 2.

A definitional interpreter is a function that simultaneously defines and applies the operational semantics of a language. In the example, the function `exec` is a candidate with the type `Config (Cmd, Config)`, i.e. it yields a configuration given a command (program) and a configuration. A definitional interpreter can also be seen to assign to every program of the language a function from configuration to configuration[2], referred to by Van Binsbergen et al. [60] as the *effect* of the program. The effects of programs can be chained by applying the definitional interpreter repeatedly to programs and using the output of one program's effect as the input of the other's. Such chaining describes the incremental style of programming of REPLs. The soundness of the approach depends on the assumption that all relevant contextual information is recorded in the configurations. The practical implications of this assumption are discussed in Section 8. By defining the definitional interpreter, the language designer determines how the execution of a program influences the execution of subsequent programs.

[2]That divergent programs cannot be represented is addressed in Section 3.

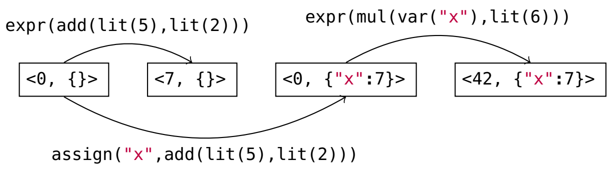


Figure 3. An execution graph for the calculator language.

Van Binsbergen et al. introduce the 'exploring interpreter' algorithm, a bookkeeping device that tracks program execution history in a graph structure, referred to as the *execution graph*, with configurations labeling nodes and programs labeling edges. Starting from the initial configuration, applications of the definitional interpreter (possibly) result in new nodes and edges added to the graph such that for every edge it holds that the source and target configurations of the edge capture the effect of the program labeling the edge.

An example execution graph is shown in Figure 3, showing the effect of three code snippets in the calculator language. The exploring interpreter algorithm supports three actions:

execute for executing programs and extending the execution graph, **revert** for changing the configuration used as input for the next execute action, and **display** for producing a structural representation of the current graph. The algorithm can be implemented generically, taking as type-level arguments the type of programs and configurations, and as (value-level) arguments a definitional interpreter and initial configuration of the correct types [11]. In this sense, the approach is (object) language-parametric.

Finally, the class of *sequential languages* is introduced for which hold that an operator $\otimes$ can be identified such that for any two syntactically valid programs p_1 and p_2 it holds that $p_1 \otimes p_2$ is a syntactically valid program whose effect is the composition of the effects of p_1 and p_2:

$$[\![p_1 \otimes p_2]\!] = [\![p_2]\!] \circ [\![p_1]\!] \qquad (1)$$

This definition states that a language is sequential if it has an operator for composing programs equivalent to chaining programs. A language with a definitional interpreter is made sequential by adding a top-level operator with its semantics given by Equation 1. A transitivity property follows stating that for every *path* in the execution graph of a sequential language, it holds that the source and target configurations capture the effects of the program formed by composing the labels of the edges of the path using $\otimes$ (in order).

In [11], Frölich and Van Binsbergen describe an implementation of the exploring interpreter algorithm and use it to evaluate the effects of certain implementation choices on the exploratory process. The authors conclude that the algorithm should support both a destructive and non-destructive variant of **revert** and that a tree view is easier to reason about since cycles are absent and there is a unique path from the root to every node – every node has a unique history.

The authors observed a distinction can be made between configuration components only used as 'output' and those influencing the execution of subsequent programs. The output components can be recorded on the labels of edges alongside the program. An example is the result field of the calculator configurations. In the next section we extend the approach to account for these suggestions.

3 Exploring Interpreter Extensions

This section describes extensions to the approach explained in Section 2. In our extended approach, a language is (optionally) defined to have output components separate from the configuration. Given a program and an input configuration, a definitional interpreter produces output and either diverges or yields an updated configuration. The definitional interpreter `exec` of the updated example has the type `tuple[Output, maybe[Config]] (Cmd, Config)` with the following definitions:

```
alias Config = tuple[Store store]; Config initial = <{}>;
alias Output = list[int]; Output no_output = [];
tuple[Output,Maybe[Config]] exec(expr(Expr e), Config c) =
  ... // produces < [], nothing() > if an unknown variable is used
```

Function `exec` yields no configuration when `eval` fails due to a reference to an unknown variable. Inspired by write-only entities in MSOS [38], output is a monoidal structure (here a list of integers) of which values can be concatenated. The transitivity property expressed at the end of Section 2 can be updated by using the monoidal operator to concatenate the output appearing as labels on edges. If a divergent program is chained with another program, the input configuration of the first is used also as the input to the second.

In our extended approach, an alternative version of the exploring algorithm is used and has been implemented as a modification to the generic algorithm of [11]. This algorithm labels the nodes of the execution graph with references rather than configurations and a mapping from references to configurations is maintained separately. The algorithm ensures the graph satisfies tree-properties by generating a fresh reference for every (successful) **execute** action. Edges are labeled with pairs of program and output. The **revert** action is destructive and a separate **jump** action is introduced as a non-destructive variant. Both receive references as argument rather than configurations. The revert action only accepts ancestors of the current node and removes only those nodes and edges that are on the path from the given ancestor r to the current node, retaining r and those nodes and edges that occur in any other paths from r. To preserve the benefits of sharing, our implementation recognizes whether a previously visited runtime state is reached by comparing configurations and maintaining sets of references pointing to the same configuration.

4 Exploratory Programming Protocol

This section introduces the Exploratory Programming Protocol (EPP), described as a sequence of TypeScript interface definitions, akin to the Language Server Protocol (LSP) [35].

The core of the EPP captures the actions of the exploring interpreter algorithm as RPC-methods and has additional methods to inspect and manipulate the execution tree. The full list of methods in the protocol is given in Table 1. The *execute*, *revert*, and *jump* methods correspond to the actions of the exploring interpreter algorithm. The *getExecutionTree*, *getTrace*, and *getPath* functions are variants of the **display** action to obtain (parts of) the execution tree in a structured format. The *meta* method gives access to meta-commands, providing language-specific services implemented in the back-end that do not involve updates to the execution tree. The remaining methods are auxiliary methods to extract information from the execution tree, such as the content of a specific configuration or a list containing all leaves.

4.1 Specification Using JSON RPC 2.0

The protocol is an instance of JSON RPC 2.0[3]. The full specification is in the supplementary material of this paper.

[3]https://www.jsonrpc.org/specification

Table 1. The methods in the exploratory programming protocol.

Method name	Description
execute	See **execute** in Section 3 and the protocol specification in §4.
revert	See **revert** in Section 3.
jump	See **jump** in Section 3.
getCurrentReference	Gets the reference labeling the current node.
getAllReferences	Returns all references used as a label.
getRoot	Returns the reference labeling the root node.
deref	Gets the configuration assigned to the given reference.
getExecutionTree	Gets the execution tree in the form of the current node a list of edges and list of nodes.
getTrace	Gets the edges representing the path from the root node to the current node.
getPath	Gets the edges representing that path between the nodes labeled by two given references.
getLeaves	Gets the references labeling the nodes without outgoing edges.
meta	Executes a meta-command without affecting the execution tree.

```
interface ExecuteRequest extends RequestMessage {
   method: "execute";
   params: ExecuteParams;
}
interface ExecuteParams {
   program: string;
}
```

Listing 1. Interface definitions for the **execute** action.

The JSON RPC 2.0 protocol defines a request object, a response object, and an error object, which are all encoded as JSON objects. A request object contains an identifier, a method name and the type capturing the parameter(s) of the method (if any). A response object contains an identifier for the request it responds to and either a result or an error. The result can be any encoded JSON object and the error object contains a unique error code, a short descriptive error message, and optional extra error data as an object.

The exploratory programming protocol is an interface between the front-end or GUI of a programming environment and an exploring interpreter serving as a back-end. The requests and response pairs of the protocol encode the actions of the exploring interpreter algorithm as JSON objects, of which we detail the **execute** specification in Listing 1. The **execute** action is encoded with a request with the method specified as "execute", and a parameter object containing a string representing the program to be executed.

As a response, the **execute** action can produce an error or a (normal) result, for which the interfaces are defined in Listing 2. The result contains the current reference from both before and after the execution, the output produced by the execution, and an optional object containing the result of post-processing the effects of the execution (discussed below). The references and the output are part of the edge added to the execution tree. The program component completing the edge is part of the request and is omitted from the response.

```
interface ExecuteResponse extends ResponseMessage {
   result?: ExecuteResult;
   error?: ExecuteError;
}
interface ExecuteResult {
   source: uinteger; // reference before execution
   target: uinteger; // reference after execution
   output: object;
   post?: object;
}
interface ExecuteError extends ResponseError {
   code: DefaultErrorCodes | ProgramParseError;
}
```

Listing 2. Interface definitions for responses to **execute**.

Following the terminology of Section 2, the *effect* of a program is the set of changes it makes to a configuration and the output it produces when successfully executed. The source, target, and output fields of an `ExecuteResult` object contain all the information necessary to compute the effects of the executed program, using a `DerefRequest` to gain access to the relevant configurations. On top of this, the optional `post` field contains any data that the back-end wishes to send to the front-end in response to an execution request by doing additional post-processing on the execution result. This can be used to compute (a summary of) the effects of a program on behalf of the front-end, as it may be more convenient to compute this information in the back-end. Such post-processing is used, for example, in the back-end for eFLINT to determine any norm violations resulting from executing a program. Finally, an **execute** operation might fail, e.g., because the program cannot be parsed (`ProgramParseError`) or the request object is invalid (`DefaultErrorCodes`).

5 Supporting Exploratory Programming

This section collects desirable features of programming environments for exploratory programming from the literature in the software engineering, human-computer interaction, and data science fields. The next section demonstrates how the EPP protocol is used to implement some of these features.

Exploratory programming is a programming style in which users (programmers) use experimentation to simultaneously find what end-result is desired, as well as the program producing that result. Exploratory programming thus requires the ability to develop a program **incrementally** and get immediate **feedback** after every submitted program (fragment). Feedback is essential to users in order to evaluate the result of the programs they submit and, if the output does not produce the desired result, users should be able to discard programs easily [51]. In particular, the user should receive enough information about the effects of the most recent execution to update their mental model of the run-time state in order to be able to predict the effects of the next program they intend to submit (e.g., see the Scrubbing Calculator [63]). From this perspective, computational notebooks do not always offer sufficient feedback [10]. And as stated by Don Norman "*poor feedback can be worse than no feedback*" [41]. User affordances are needed to inform about **program state** (the executed program (fragments)) and **run-time state** (the context in which the next execution will take place).

Micro-Versioning. Users should be able to work with different versions of both code and (intermediate) results [3, 14, 21, 65]. As such, users can better understand the design space and make better coding decisions. However, fine-grained (sub-file level) support for versioning, referred to as **micro-versioning** by Hiroaki et al. [36], is not common in present-day programming tools. From the interaction-design perspective, micro-versioning is challenging because users have to cognitively maintain multiple representations of code and the running program [14, 65]. In other words, users have to maintain multiple mental models of program and run-time states and which program resulted in which run-time state.

Software engineers use version control systems like Git or Subversion for versioning large-scale software projects. However, version control systems operate at the project and file level instead of the level of program fragments making them insufficient for micro-versioning as described above [20].

Experimentation and Modification. Systems that are difficult to modify are referred to as systems of high *viscosity* and are not suited for exploratory programming [3, 13]. Exploratory programming requires quick and easy creation and modification of program fragments while editing or after execution, without noticeable overhead [7, 16, 31].

Notebooks typically allow users to modify and re-execute existing code cells, but do not keep track of previous version of a code cell. This may leave a notebook in an inconsistent state in the sense that the contents of the code cells no longer respect (data) dependencies or give a different result when executed from scratch [10]. Users can also decide to copy cell contents to a new code cell [29]. This strategy ensures that program state remains consistent with run-time state, but results in an ever-growing program that records earlier unsuccessful experiments. As a result, users tend to start from scratch to avoid the aforementioned situations. Code cloning is a common practice in software development [49] and it is even more common and noticeable in exploratory programming environments [54]. Trial-and-error causes users to copy-paste code snippets with modifications to explore alternatives [3, 16, 65]. Especially non-expert users copy-paste code snippets and start tweaking them to understand their semantics and to adapt them to achieve their goals [31]. Users require a mechanism to display the **history** of the different commands they have executed in their sessions with a limited number of actions and be able to interact the alternatives they have created [16, 21, 33, 36, 65]. As discussed later, our experimental front-end supports **backtracking** with unlimited levels of undo/redo and branching explicitly, as suggested by Hauswirth and Azadmanesh [15].

Managing Alternatives. The exploratory style of programming may result in a collection of alternative versions of a program, which can grow rapidly with little organization [10]. It may be hard for users to keep track of the alternatives and the intermediate results they achieved throughout their explorations. Users should be able to **browse** through alternative program states, using an efficient representation of the possibly large number of alternatives. It should be possible to **compare** alternatives, both with respect to their program state and (intermediate) run-time state(s), to select the best candidate for further exploration. The ability to **search** through alternatives aids selection even further [10].

The documentation cells of notebooks enable literate programming [27] for explaining design choices and functionality. In an exploratory setting it should also be possible to **document** the exploration process itself. This gives users insights into the thinking that went into different exploration attempts, thereby supporting users in understanding (intermediate) results and attempting further explorations.

Outcomes should be easy to share with other users when an exploration session concludes. The ability to **reuse** code is important in software engineering, and **reproducability** is an important principle in (data) science. Jupyter notebooks are saved in a textual format that can be subjected to version control, shared easily with other users and rendered as HTML for online publication. An exploratory programming environment that supports the exploration of alternative programs has additional requirements. In particular, it should be possible to share only part of the exploration, i.e. only those alternatives having produced desirable results. But since the exploration process itself may contain meaningful information, e.g., about documented implementation decisions and false starts, sharing multiple alternatives should be an option. Ideally, the user has the ability to choose freely which alternatives to make available for sharing in a form that preserves the features mentioned earlier: (micro-)versioning, feedback, browse, compare, search, and document(ation).

6 An Experimental Notebook Front-end

This section describes the features of a GUI interface which has been designed to experiment with novel features for exploratory programming in a notebook-like environment. The front-end communicates with a back-end via the EPP protocol. In particular, we use the front-end to demonstrate how certain features discussed in the previous section can be realized on top of the EPP (references to features appear in **bold**). The architecture on which we performed our experiments is described in the next section. A thorough design and evaluation of GUI components is part of future work. The features of the front-end are generic in the sense that they are not designed for a specific object language and behave similarly across languages. The artifact submitted alongside this paper includes the prototypes for the Idris and eFLINT languages using this front-end.

The main screen of the front-end, in this case for Idris, is shown in Figure 4. As the front-end is based on the EPP, it naturally supports **backtracking** and jumping to previous program states (**history**). Rather than visualizing the execution tree such as in [33, 60], the front-end shows a single 'execution trace' corresponding to the path in the tree from the root node to the node representing the current program state (center component). For each edge in the trace, the executed program is shown together with its output[4] (**feedback**) and a button to revert to the state prior to that execution. The user is able to switch between execution traces by selecting the 'Switch trace' button on the right-hand side, associated with 'head nodes' and 'tagged node' (**browse**). Head nodes correspond to the leaves of the underlying execution tree, tagged nodes are selected by the user, e.g. to record states that have achieved interesting intermediate results (**document**). The head and tagged nodes are shown on the right-hand side of the screen with their identifier (the reference labeling the node of the execution tree) and the program executed to produce that node. The left-hand side contains code cells, output cells and documentation cells as is common in computational notebooks.

The front-end supports **incremental** program execution through the execution of code cells in two ways. Firstly, the notebook (left-hand side) component can be extended with new code cells for execution, and existing code cells can be modified and re-executed. The 'Actions' button attached to a code cell reveals, among others, a 'Revert' button and an 'Execute' button. To support **micro-versioning** and to better keep tracking of execution **history**, code cells in the notebook are associated with one or more edges of the execution tree, always showing one pair of 'Previous state' and 'Output state', together with the output labeling the edge (**feedback**). Under 'Actions', the user can switch between different executions of the same cell. Secondly, the code cells that make up the execution trace (center) can be modified and re-executed. The "modify and re-execute" button attached to these cells makes it possible to execute the modified program in the state prior to its original execution, using a jump, and to subsequently re-execute all code fragments whose effects were undone by the jump, resulting in a new branch in the execution tree. The new branch is represented by a new head node on the right-hand side of the interface and is shown as the current execution trace. These features make it possible to experiment by creating alternative explorations as (minor) modifications of existing explorations (**micro-versioning**). Note that a modification to the cell may be such that the subsequent code fragments are no longer type correct, in which case errors are produced as they would normally.

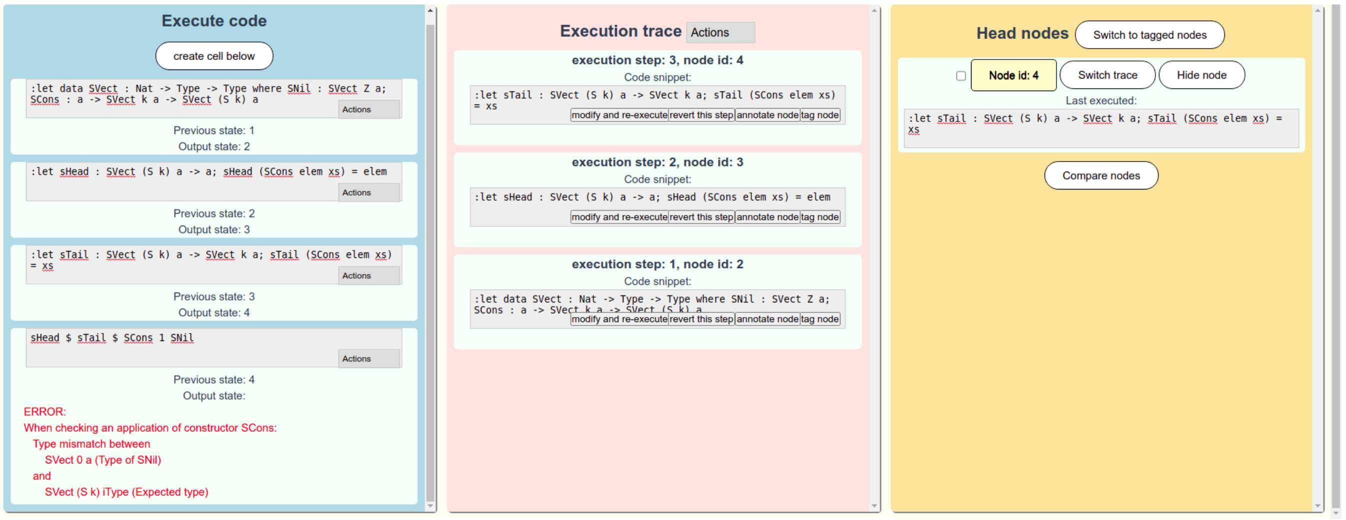
The execution trace shows a **program state** that is consistent with the current **run-time state**. And as discussed, the notebook component is capable of keeping track of multiple versions of code cells. However, executing code cells from within the execution trace may cause inconsistency between the execution trace and the narrative of the notebook on the left-hand side. For this purpose, it is possible to 'migrate' the execution trace to the notebook by creating code cells with the context of the programs in the trace available under 'Actions'. In this case, the annotations added to code cells in the execution trace (maintained by the front-end) can be turned into documentation cells. They are also used to add documentation to tagged cells in order to document the exploration process (**document**).

The head and tagged nodes appearing on the right-hand side can be selected (checkbox) for comparison. Clicking the 'Compare nodes' button opens a pop-up such as the one shown in Figure 5. The view has tabs for comparing configurations – summarizing **run-time state** – traces, and the annotations attached to traces (**compare**). This way, traces, individual configurations, and annotations can be placed side-by-side for comparison. Structural diff algorithms could be applied to show the difference to the user, but this feature is not yet part of our experimental front-end.

A **search** field is also available to filter the contents of a trace/configuration (Figure 5). The generic implementation of this feature performs a textual search on the underlying HTML. More sophisticated language-specific search options could be realized as well. For example, a programmer might like to search for configurations that satisfy some property written as a Boolean expression in the object language. In the next section we discuss how the Idris version of this front-end features a specialized version of search with which a user can search for occurrences and declarations of a variable.

The front-end is able to export and import execution trees, using the 'getExecutionTree' method of the EPP. This way

[4]No output is produced by the declarations in Figure 4.



```
:let data SVect : Nat -> Type -> Type where SNil : SVect Z a;
SCons : a -> SVect k a -> SVect (S k) a
```

Previous state: 1
Output state: 2

```
:let sHead : SVect (S k) a -> a; sHead (SCons elem xs) = elem
```

Previous state: 2
Output state: 3

```
:let sTail : SVect (S k) a -> SVect k a; sTail (SCons elem xs)
= xs
```

Previous state: 3
Output state: 4

```
sHead $ sTail $ SCons 1 SNil
```

Previous state: 4
Output state:

```
ERROR:
When checking an application of constructor SCons:
  Type mismatch between
    SVect 0 a (Type of SNil)
  and
    SVect (S k) iType (Expected type)
```

execution step: 3, node id: 4

```
:let sTail : SVect (S k) a -> SVect k a; sTail (SCons elem xs)
= xs
```

execution step: 2, node id: 3

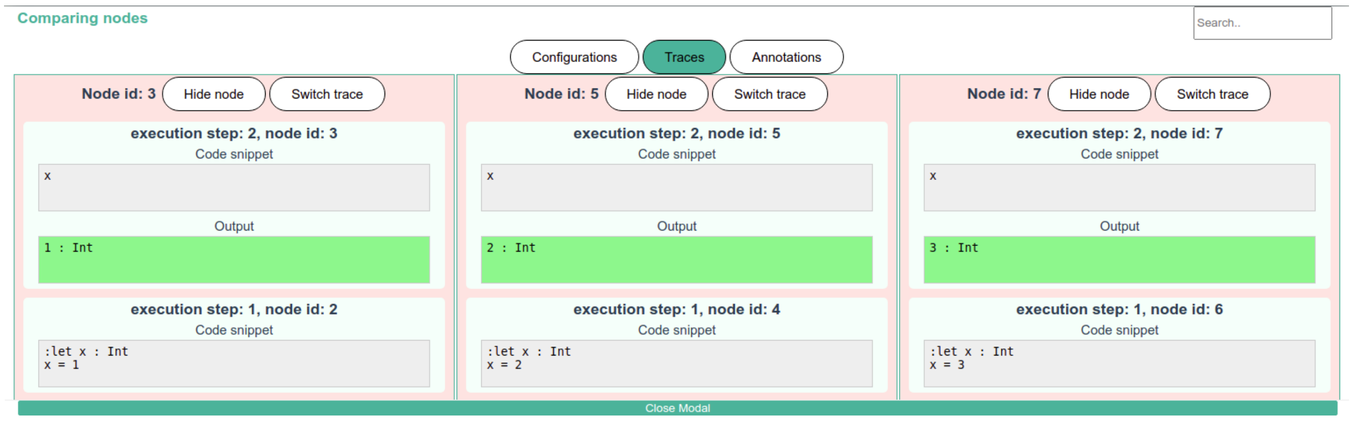
```
:let sHead : SVect (S k) a -> a; sHead (SCons elem xs) = elem
```

execution step: 1, node id: 2

```
:let data SVect : Nat -> Type -> Type where SNil : SVect Z a;
SCons : a -> SVect k a -> SVect (S k) a
```

```
:let sTail : SVect (S k) a -> SVect k a; sTail (SCons elem xs) =
xs
```

Figure 4. An experimental notebook interface with various generic exploratory programming features used with the Idris language. The dependently typed nature of Idris is shown by performing *sHead* on an empty vector, resulting in a type error.



execution step: 2, node id: 3

```
x
```

```
1 : Int
```

execution step: 1, node id: 2

```
:let x : Int
x = 1
```

execution step: 2, node id: 5

```
x
```

```
2 : Int
```

execution step: 1, node id: 4

```
:let x : Int
x = 2
```

execution step: 2, node id: 7

```
x
```

```
3 : Int
```

execution step: 1, node id: 6

```
:let x : Int
x = 3
```

Figure 5. Screenshot of the Idris prototype, showing a pop-up in which traces are compared.

exploration sessions can be shared with other users and subjected to version control (**reuse** and **reproducibility**). Individual traces can also be exported using 'getTrace' making it possible to share only certain desirable traces. Additional export functionality could be developed on top of these (and other) methods of the protocol too, for example, to export exactly those traces in the tree ending in tagged nodes.

7 Reusable Architecture Implementation

This section describes the design and implementation of an architecture that enables research into generic or language-specific (UI) features for exploratory programming. Based on the EPP, the architecture demonstrates that the EPP is language-parametric in that it can be used for object languages for which a definitional interpreter is available (implemented in the back-end host language). The architecture is visualized in Figure 6. Given a choice of host languages for the front- and back-end, some components of the architecture are reusable across prototypes, as indicated by the dashed components. We explain how we used the protocol to implement several prototype programming environments for different object languages, emphasizing the connection between reusable components and language- or UI-specific components. The architecture has been used to implement prototype notebooks and REPLs for Idris [5], MiniJava [1, 8], eFLINT [57] and Funcons-beta [58]. The Idris and Funcons-beta prototypes are especially interesting as they are built on top of existing interpreters developed without anticipating their usage with the EPP. Other prototypes, such as the QL prototype described in Section 1 preceded the work presented in this paper and inspired the formulation of the protocol as well as the design and implementation of the architecture. The implemented prototypes, as instances of

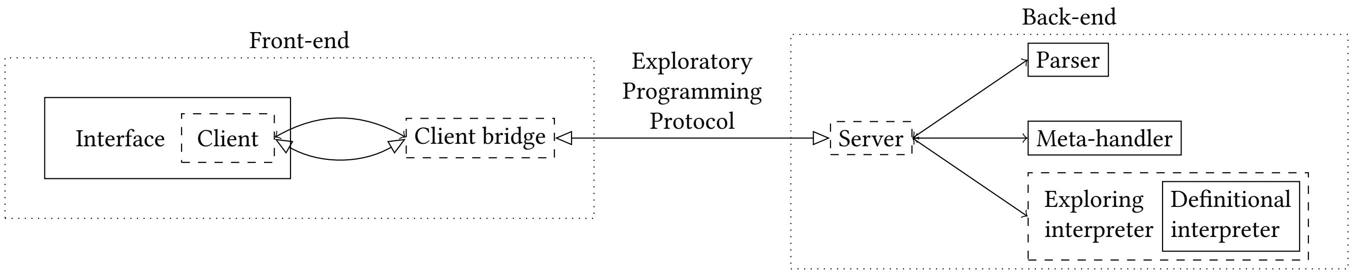


Figure 6. The architecture designed for prototyping programming environments with the Exploratory Programming Protocol. Rectangles with dashed lines indicate reusable components given a choice of host languages for the front- and back-end. Solid rectangles are language- or environment-specific components. Arrows with open triangles depict network communication, whereas solid arrows depict function application. Both connection methods are used for the client and client bridge.

the architecture, are available as part of the supplementary material of this paper.

7.1 Back-end

The back-end consists of a **server** parameterized by the following (object) language-specific components: a parser, a meta-handler, and a definitional interpreter. The definitional interpreter is used to instantiate a generic exploring interpreter, which maintains the execution tree. The server transforms a message from the protocol into operations on the three components. It then takes the result of these operations and transforms them into a message according to the protocol and sends it to the front-end. For example, execute requests are realized via the parser and the definitional interpreter by first parsing the input string with the given parser and then invoking the exploring interpreter.

Our prototypes are based on a reusable Haskell implementation of the server and exploring interpreter components. The latter is a modification of the implementation of [11] to account for the extensions in Section 3. The implementation of the execute method within the Haskell server is shown in Listing 3 (simplified for clarity). The request object is parsed as a JSON object. If the request is not correctly formatted, the `InvalidParameters` error is returned. Otherwise, the parser is applied to the program field of the request, returning an error (`Left err`) or a parsed program (`Right prog`). When parsing is successful, the exploring interpreter executes the program, resulting in an extended execution tree and optional output. The source and target labels (references) of the new edge are part of the result, together with the output and the result of any post-processing.

The **parser** is a language-specific component with the signature: *String -> c -> Either String p*, where *c* represents the configurations of the language and *p* the programs. The parser yields either a program or an error, and it has access to the current configuration. Access to the current configuration can be useful for context-sensitive parsing, e.g., Idris

```
execute ::Value -> ErrorT ErrorMessage (EIP p IO c o) Value
execute v = case (fromJSON v) ::Result ExecuteParams of
  (Error e) -> throwError invalidParams
  (Success v_new) -> do
    case parse $ program (v_new ::ExecuteParams) of
      Right prog -> do
        (ex_new, output) ⟵Ex.execute prog ex
        return $ toJSON $ ExecuteResult
          { source = Ex.currRef ex
          , target = Ex.currRef ex_new
          , output = toJSON output
          , post = postExecute ex ex_new output }
      Left err -> throwError ErrorMessage
        { code = programParseErrorCode
        , message = "Supplied program is invalid"
        , error_data = toJSON err }
```

Listing 3. Simplified Haskell fragment of the implementation of **execute** in the back-end server.

allows dynamic extensions of syntax. When the parse is unsuccessful, the parser can provide an error message sent to the front-end as part of the the error object.

Via the **meta-handler**, a back-end can deliver additional features. A meta-handler has the signature: *Value -> Explorer p m c o -> m Value*. The handler receives a parameter of the request (a JSON value), the current exploring interpreter, and returns a JSON value. The meta-handler has access to the exploring interpreter to support reading the execution tree. In our Idris prototype, we use meta-commands to provide semantics-based search through the execution tree. This search finds all leaves in which an identifier occurs before searching for all nodes where the identifier was declared.

The **definitional interpreter** implements the operational semantics of the language and has the following type signature: *p -> c -> m (Maybe c, o)*, where *p* are the programs, *c* configurations, *o* is the monoidal output component, and *m* is an arbitrary monad in which the interpreter can execute. The signature is general in the sense that many languages can have an interpreter implemented according to the required signature, or that an existing interpreter can be adapted to adhere to the signature. For example, the Idris prototype uses a definitional interpreter implemented as a wrapper around an existing interpreter for the language [6].

Reflections on Reuse. The prototypes we developed as instances of the architecture use Haskell implementations of parsers, definitional interpreters, and meta-handlers. Both the server and the exploring interpreter components are language-parametric, and therefore only needed to be implemented once. The server and exploring interpreter need to be re-implemented in a different host language to use the protocol and architecture for object languages implemented in that host language. However, this (hypothetical) novel back-end can be combined with existing front-ends, as it relies on the language-agnostic EPP for its communication.

7.2 Front-end

The front-end is divided into two parts: an interface that extends a reusable client and a bridge that connects the front-end to the back-end.

The **client** provides an API for the interface developer abstracting over communication details. This is achieved by defining the client as an abstract class consisting of concrete methods for performing EPP requests and abstract methods for handling the EPP responses. The concrete methods are implemented once and for all within the client and are generic. These methods assign a unique reference to every request and store the request to be later matched with a response. After receiving a response from the client bridge, the client calls the corresponding request handler method. These handler methods are language-specific and must be implemented for every interface. For example, in a prototype for the eFLINT language, an execute action is performed as follows (see Listing 4). A button click triggers the execution of a code cell by calling the handler of the click event *doExecute* which calls the method *execute* of the client by providing the `ExecuteParams` of the request. When the response arrives, the client calls the *onExecute* method with the original request and the response as arguments. The *onExecute* method first determines if the request was successful or not. If the request was successful, the method calls the `showViolations` method to display any violations to the user when any violations are discovered by the back-end's post-processing.

The **client bridge** is an adapter, translating messages from the protocol used between the client and the client bridge into messages of the EPP, and vice versa. This layer of indirection makes it possible to support a wide variety of front-end implementations separate from back-ends.

Reflections on Reuse. For our prototypes we have two implementations of the client component (hence the two arrows between client and client bridge in Figure 6): one implementation uses the WebSocket protocol as the communication format between the client and client bridge and the

```
doExecute(input: string) {
  this.execute(new ExecuteParams(input);
}
...
onExecute(req: ExecuteRequest, resp: ExecuteResponse) {
 if (resp.error) {
      this.handleExecuteError(req, resp);
      return;
 }
 showViolations(resp.post);
 ... // additional code making changes in the front-end
}
```

Listing 4. TypeScript code that shows part of an interface implementation for the eFLINT language.

other uses a native UI, implemented using Python and the Tk interface. In the latter implementation, the client and client bridge are connected directly via function calls. Both front-end implementations are used to develop prototypes on the same Haskell back-end. In general, any implementation of the client (bridge) component can be used in combination with any implementation of the server component.

The **interface** component can be implemented with both features and widgets that are generic and specific to a certain object language. Features can be developed on top of the generic part of the protocol, e.g. executing code in code cells, displaying execution traces, jumping to previous run-time states, etc. Such features are reusable across languages, reducing the workload for language engineers and providing a common experience for programmers switching between languages. On the other hand, a more tailored experience can be offered to programmers with features which are designed specifically for a particular object language, e.g., using post-processing and meta-handlers. With our architecture we can combine generic and language-specific features and replace generic features with specialized variants when available.

One way specialization is achieved is by making the implementation of a feature parametric such that language-specific behavior can be provided as an argument. For example, a variable watcher [33] – showing the assignments to variables in the current run-time state – can be implemented such that a function is given as an argument that extracts variable assignments from a configuration. A different argument is used for different object languages as each language has its own notion of configuration and approach to keeping track of assignments. Other examples are output cells and visualizations of the execution history when they include information extracted from configurations.

Another approach to specialization is overriding or extending a generic implementation of a feature. The default, generic implementation of the search functionality of our experimental front-end is realized in a text-based fashion by searching the DOM-rendering of the trace. In the Idris prototype, this implementation is replaced with a semantics-based search using the meta-handler for Idris mentioned earlier. The semantic search can be used to find only those code cells in a trace in which some variable x is declared or used, whereas with text-based search all occurrences of the letter 'x' will be found. Another example is for the eFLINT language, in which code cells have been extended with an indicator of any norm violations caused by executing the code cell. When the programmer clicks on the sign, the violations introduced by the program are shown in a modal dialog. In the search example, specialization was realized using the meta-handler for Idris. In the eFLINT example, the specialization was realized by the back-end applying post-processing to extract violations from the configuration produced by executing a program. These examples are specializations of otherwise generic features of an experimental notebook interface. Note, however, that this experimental front-end is but one implementation of the interface component of our architecture and that the above discussion explains reuse within that component. We have also developed prototypes using other interface implementations. For example, we have implemented a REPL running as a web interface. Since the REPL is developed using generic parts of the protocol, we can use the REPL for the other object languages without modifying the back-end. Another implementation in Python uses a native UI with a Tk interface.

8 Discussion

From the previous sections we conclude that the protocol offers benefits to the software language engineering process of building programming environments that support exploratory programming. We can use the protocol and architecture to experiment with the design of exploratory programming features with relative ease by reusing components. In Section 6 we provided evidence to the claim that the EPP supports interesting exploratory programming scenarios by relating features of a prototype to scenarios discussed in the literature. The thorough design and evaluation of GUI elements and features for exploratory programming is part of future work.

Applicability. The main limitation of our approach is that we rely on the availability of definitional interpreters and parsers for the object languages with which we wish to experiment. However, when a definitional interpreter is available, existing (generic) interfaces are immediately applicable to the new object language. Language-specific GUI elements, meta-handler functionality or post-processing steps can then be added on a by-need basis. The Idris and Funcons-beta examples show that existing interpreters can be reused, even when they have been developed without anticipating the EPP. To embed the Idris interpreter of [6] into our architecture, only four lines of Haskell code where needed to map errors to output. Adapting the interpreter of Funcons-beta [59], required around 50 lines of Haskell to extend the language to a sequential variant, following the methodology proposed in [60], and propagating bindings between funcon terms.

As indicated in [60], the class of languages to which the approach can be applied contains all languages that can have their semantics expressed as a (possibly partial) transition function. This class contains real-world, large-scale, deterministic programming languages, as is demonstrated by the body of literature on big-step, small-step and natural semantics [2, 19, 37, 39, 44] and does not necessarily exclude languages with non-deterministic aspects when these aspects can be captured algebraically [64]. However, implementing such a transition function as an efficient, production-ready interpreter maintaining explicit representations of configurations is another matter. Comparing alternative development strategies for such interpreters and demonstrating the practicality of the EPP in real-world environments is left as future work. The goal of this work has been to create an environment for experimenting with user interfaces and functionality for exploratory programming, without having run-time or space requirements as a limiting factor on the design space being explored. We intend to capitalize with future collaborations in the spirit of 'PL and HCI: better together' [9].

Object Languages. Our approach is particularly useful in the context of DSLs, since DSL engineers make different trade-offs regarding performance. Fast prototyping and design iteration with stakeholders is often more valuable than raw speed. In that sense, this work adds exploratory tooling for 'free', to the language workbench's tool box. Several of the prototypes we developed are indeed for DSLs (eFLINT and QL). The design of object languages plays an essential role in the support for exploratory programming. DSLs capture abstractions tailored to specific domains, and programming environments can offer such high-level abstractions (and others) as widgets [3, 55] which can yield better and more powerful explorations [52]. The rendering of the QL form (Figure 1) is an example of such a language-specific widget, with modifications being reified as code [24, 66].

The exploratory programming protocol makes no assumptions about whether the object language is statically or dynamically typed. Exploration in statically typed languages is interesting because type-checking is expected to be performed on individual program fragments. This means that typing information needs to propagate between the (dynamic) execution of fragments and that, in some sense, the distinction between static analyses and dynamic evaluation becomes blurred. Figure 4 shows how a type error produced by the interpreter for Idris – a dependently typed language – is presented as output. Similar to offering flexibility in typing, exploratory programming can also significantly benefit from being able to submit partial programs with holes and receiving feedback on these holes [12, 42].

Debugging programs is another important aspect of exploratory programming [7, 28], Traditional debugging requires users to switch between a source code and a debugging view, hindering users from having a clear picture of the run-time state, and preventing them to seamlessly continue experimenting [17]. Live programming helps users understand and comprehend their programs by giving immediate feedback about the program state after a change to the source code [45]. This is, for instance, demonstrated in our QL prototype. In another experiment, we treat stepwise debugging as a matter of language design. The methodology of [60] can be extended to incorporate stepwise debugging features, assuming a 'stepwise interpreter' is available. This achieved by adding elementary debugging constructs such as *debug*(e), for some expression e, *step*, and *continue* as phrases to the object language. The intermediate results of steps are recorded in the execution tree, enabling jumping to an earlier point in a debugging session, reminiscent of omniscient debugging [4, 32]. The QL and Funcons-beta prototypes support stepwise debugging in this way.

Some computational notebooks enable polyglot programming in which multiple object languages are used simultaneously [40, 43, 53]. Being able to apply multiple languages within the same exploration session is considered desirable [7]. We are performing experiments to determine how to enable polyglot programming within our approach.

9 Related Work

Exploratory programming is becoming increasingly important, especially as the number of end-users is outgrowing the number of professional programmers [48]. There is a need for better tools and languages aimed at end-users. In this direction, computational notebooks (e.g., Jupyter, ObservableHQ, Apache Zeppelin, and Google Collaboratory) have become an interesting and popular solution used by end-users when they need to work with code, prose, and interactive results. However, as found in the literature, these programming environments have some limitations, especially for common exploratory programming tasks [10, 16, 20, 22, 23, 46, 50].

Everyday exploratory programming tasks require that users have to deal with different explorations [3, 14, 16]. For instance, Juxtapose [14] is a tool for managing different alternatives across source code and execution environments. It allows users to execute alternatives in parallel and display their results in the same window. This is a crucial task for exploratory programming not supported by popular notebooks. Juxtapose also generates a control interface that allows users to manipulate application parameters through sliders. Exploratory programming activities require support for versioning [23]. However, given that exploratory programming often relies on incremental program development, traditional software versioning systems are too complex and challenging to use for end-users. Therefore, Micro-versioning [36] is an interesting approach for versioning partial programs and results in exploratory programming environments.

Another critical aspect is the diversity of tools for exploratory programming. As the number of users increases, the number of exploratory programming environments is also increasing [30, 62]. This offers benefits to users; however, language developers need to offer support for different platforms, which is a cumbersome and expensive task. To address this problem, some protocols (e.g., the Language Server Protocol [35] and Debug Adapter Protocol [34]) have been defined to standardize communication between tools and languages so that language and tool developers can reuse a single implementation across platforms. For instance, LSP enables the communication between code editors and languages to offer different IDE services (e.g., auto-completion, go to definition, etc.). However, LSP does not formalize an API to manage the execution of programs. In Section 4, we present a first approach towards defining a language-independent protocol that considers the execution step, primarily to support exploratory programming scenarios.

10 Conclusion

We have presented a generic protocol and a reusable architecture for programming environments supporting exploratory programming, a style of programming characterized by prototyping, versioning, and various forms of experimentation. The protocol is generic in that it can be used for a large class of object languages and can be implemented in various host languages. The architecture enables us to experiment with novel features for exploratory programming in environments such as computational notebooks and REPLs. The prototypes developed in our experiments demonstrate that the protocol can be used to deliver features for many exploratory programming scenarios discussed in the literature. The next step in our research is to design, implement, and evaluate generic and language-specific user-interface components for exploratory programming, by taking advantage of the protocol and architecture presented in this paper.

Acknowledgments

This work has been partially supported by the Kansen Voor West EFRO project (KVW00309) *AMdEX Fieldlab* and the province of Noord-Holland, and has been executed as part of the Agile Language Engineering collaboration[5].

References

[1] Andrew W. Appel and Jens Palsberg. 2003. *Modern Compiler Implementation in Java* (2nd ed.). Cambridge University Press.

[2] Egidio Astesiano. 1991. Inductive and Operational Semantics. In *IFIP State-of-the-Art Reports, Formal Descriptions of Programming Concepts*, E.J. Neuhold and M. Paul (Eds.). Springer, 51–136.

[5] http://gemoc.org/ale/

[3] Mary Beth Kery and Brad A. Myers. 2017. Exploring exploratory programming. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 25–29. https://doi.org/10.1109/VLHCC.2017.8103446

[4] Erwan Bousse, Dorian Leroy, Benoît Combemale, Manuel Wimmer, and Benoit Baudry. 2018. Omniscient debugging for executable DSLs. *Journal of Systems and Software* 137 (2018), 261–288. https://doi.org/10.1016/j.jss.2017.11.025

[5] Edwin Brady. 2013. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming* 23, 5 (2013), 552–593. https://doi.org/10.1017/S095679681300018X

[6] Edwin Brady and Niklas Larsson. 2021. Idris on Hackage. (2021). https://hackage.haskell.org/package/idris [Online, accessed 27 September 2022].

[7] Joel Brandt, Philip J. Guo, Joel Lewenstein, Mira Dontcheva, and Scott R. Klemmer. 2009. Writing Code to Prototype, Ideate, and Discover. *IEEE Software* 26, 5 (2009), 18–24. https://doi.org/10.1109/MS.2009.147

[8] João Cangussu, Jens Palsberg, and Vidyut Samanta. 2002. The MiniJava Project. https://www.cambridge.org/us/features/052182060X. (2002). [Online, accessed 12 October 2020].

[9] Sarah E. Chasins, Elena L. Glassman, and Joshua Sunshine. 2021. PL and HCI: Better Together. *Commun. ACM* 64, 8 (jul 2021), 98–106. https://doi.org/10.1145/3469279

[10] Souti Chattopadhyay, Ishita Prasad, Austin Z. Henley, Anita Sarma, and Titus Barik. 2020. What's Wrong with Computational Notebooks? Pain Points, Needs, and Design Opportunities. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems (CHI '20)*. ACM, 1–12. https://doi.org/10.1145/3313831.3376729

[11] Damian Frölich and L. Thomas van Binsbergen. 2021. A Generic Back-End for Exploratory Programming. In *The 22nd International Symposium on Trends in Functional Programming (TFP 2021) (LNCS)*, Vol. 12834. Springer. https://doi.org/10.1007/978-3-030-83978-9_2

[12] Alex Gerdes, Bastiaan Heeren, Johan Jeuring, and L. Thomas van Binsbergen. 2017. Ask-Elle: an Adaptable Programming Tutor for Haskell Giving Automated Feedback. *Int. J. Artif. Intell. Educ.* 27, 1 (2017), 65–100. https://doi.org/10.1007/s40593-015-0080-x

[13] T.R.G. Green. 1990. Chapter 2.2 - Programming Languages as Information Structures. In *Psychology of Programming*, J.-M. Hoc, T.R.G. Green, R. Samurçay, and D.J. Gilmore (Eds.). Academic Press, 117–137. https://doi.org/10.1016/B978-0-12-350772-3.50013-6

[14] Björn Hartmann, Loren Yu, Abel Allison, Yeonsoo Yang, and Scott R. Klemmer. 2008. Design as Exploration: Creating Interface Alternatives through Parallel Authoring and Runtime Tuning. In *Proceedings of the 21st Annual ACM Symposium on User Interface Software and Technology (UIST '08)*. ACM, 91–100. https://doi.org/10.1145/1449715.1449732

[15] Matthias Hauswirth and Mohammad Reza Azadmanesh. 2017. The Entangled Strands of Time in Software Development *(PX/17.2)*. ACM, 11–16. https://doi.org/10.1145/3167107

[16] Andrew Head, Fred Hohman, Titus Barik, Steven M. Drucker, and Robert DeLine. 2019. *Managing Messes in Computational Notebooks.* ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/3290605.3300500

[17] Jane Hoffswell, Arvind Satyanarayan, and Jeffrey Heer. 2018. Augmenting Code with In Situ Visualizations to Aid Program Understanding. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems (CHI '18)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/3173574.3174106

[18] MathWorks Inc. 2019. MATLAB Live Editor. https://nl.mathworks.com/products/matlab/live-editor.html. (2019). https://nl.mathworks.com/products/matlab/live-editor.html [Online, accessed 16 July 2021].

[19] Gilles Kahn. 1987. Natural Semantics. In *Proceedings of the 4th Annual Symposium on Theoretical Aspects of Computer Science*. Springer-Verlag, 22–39. https://doi.org/10.1007/BFb0039592

[20] Mary Beth Kery, Amber Horvath, and Brad Myers. 2017. *Variolite: Supporting Exploratory Programming by Data Scientists.* ACM, New York, NY, USA, 1265–1276. https://doi.org/10.1145/3025453.3025626

[21] Mary Beth Kery, Bonnie E. John, Patrick O'Flaherty, Amber Horvath, and Brad A. Myers. 2019. Towards Effective Foraging by Data Scientists to Find Past Analysis Choices. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems (CHI '19)*. ACM, New York, NY, USA, 1–13. https://doi.org/10.1145/3290605.3300322

[22] Mary Beth Kery and Brad A. Myers. 2018. Interactions for Untangling Messy History in a Computational Notebook. In *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 147–155. https://doi.org/10.1109/VLHCC.2018.8506576

[23] Mary Beth Kery, Marissa Radensky, Mahima Arya, Bonnie E. John, and Brad A. Myers. 2018. *The Story in the Notebook: Exploratory Data Science Using a Literate Programming Tool.* ACM, New York, NY, USA, 1–11. https://doi.org/10.1145/3173574.3173748

[24] Mary Beth Kery, Donghao Ren, Fred Hohman, Dominik Moritz, Kanit Wongsuphasawat, and Kayur Patel. 2020. mage: Fluid Moves Between Code and Graphical Work in Computational Notebooks. *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology* (Oct 2020). https://doi.org/10.1145/3379337.3415842

[25] Paul Klint, Tijs van der Storm, and Vinju Jurgen. 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

[26] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian E. Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica B. Hamrick, Jason Grout, Sylvain Corlay, Paul Ivanov, Damián Avila, Safia Abdalla, Carol Willing, and et al. 2016. Jupyter Notebooks - a publishing format for reproducible computational workflows. In *Positioning and Power in Academic Publishing: Players, Agents and Agendas, 20th International Conference on Electronic Publishing, Göttingen, Germany, June 7-9, 2016*. 87–90. https://doi.org/10.3233/978-1-61499-649-1-87

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

[28] Amy J. Ko, Robin Abraham, Laura Beckwith, Alan Blackwell, Margaret Burnett, Martin Erwig, Chris Scaffidi, Joseph Lawrance, Henry Lieberman, Brad Myers, Mary Beth Rosson, Gregg Rothermel, Mary Shaw, and Susan Wiedenbeck. 2011. The State of the Art in End-User Software Engineering. *ACM Computer Surveys* 43, 3 (April 2011). https://doi.org/10.1145/1922649.1922658

[29] Andreas P. Koenzen, Neil A. Ernst, and Margaret-Anne D. Storey. 2020. Code Duplication and Reuse in Jupyter Notebooks. In *2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 1–9. https://doi.org/10.1109/VL/HCC50065.2020.9127202

[30] Sam Lau, Ian Drosos, Julia M. Markel, and Philip J. Guo. 2020. The Design Space of Computational Notebooks: An Analysis of 60 Systems in Academia and Industry. In *2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 1–11. https://doi.org/10.1109/VL/HCC50065.2020.9127201

[31] Sam Lau, Sruti Srinivasa Srinivasa Ragavan, Ken Milne, Titus Barik, and Advait Sarkar. 2021. TweakIt: Supporting End-User Programmers Who Transmogrify Code. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems (CHI '21)*. ACM, New York, NY, USA. https://doi.org/10.1145/3411764.3445265

[32] Bil Lewis. 2003. Debugging Backwards in Time. *Computing Research Repository* cs.SE/0310016 (2003). http://arxiv.org/abs/cs/0310016

[33] Mauricio Verano Merino, L. Thomas van Binsbergen, and Mazyar Seraj. 2022. Making the Invisible Visible in Computational Notebooks. In *2022 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 1–3. https://doi.org/10.1109/VL/HCC53370.2022.9833148

[34] Microsoft. 2018. Debug Adapter Protocol. (2018). https://microsoft.github.io/debug-adapter-protocol/

[35] Microsoft. 2018. Language Server Protocol. (2018). https://microsoft.github.io/language-server-protocol [Online, accessed 16 July 2021].

[36] Hiroaki Mikami, Daisuke Sakamoto, and Takeo Igarashi. 2017. Micro-Versioning Tool to Support Experimentation in Exploratory Programming. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems (CHI '17)*. ACM, 6208–6219. https://doi.org/10.1145/3025453.3025597

[37] Robin Milner, Mads Tofte, and David MacQueen. 1997. *The Definition of Standard ML*. MIT Press.

[38] Peter D. Mosses. 2004. Modular Structural Operational Semantics. *Journal of Logic and Algebraic Programming* 60–61 (2004), 195–228. https://doi.org/10.1016/j.jlap.2004.03.008

[39] Peter D. Mosses. 2004. Modular Structural Operational Semantics. *Journal of Logic and Algebraic Programming* 60–61 (2004), 195–228. https://doi.org/10.1016/j.jlap.2004.03.008

[40] Fabio Niephaus, Eva Krebs, Christian Flach, Jens Lincke, and Robert Hirschfeld. 2019. PolyJuS: A Squeak/Smalltalk-Based Polyglot Notebook System for the GraalVM. In *Proceedings of the Conference Companion of the 3rd International Conference on Art, Science, and Engineering of Programming (Programming '19)*. ACM, New York, NY, USA. https://doi.org/10.1145/3328433.3328434

[41] Donald A. Norman. 2002. *The Design of Everyday Things*. Basic Books, Inc.

[42] Cyrus Omar, Ian Voysey, Michael Hilton, Jonathan Aldrich, and Matthew A. Hammer. 2017. Hazelnut: A Bidirectionally Typed Structure Editor Calculus. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL '17)*. ACM, New York, NY, USA, 86–99. https://doi.org/10.1145/3009837.3009900

[43] Bo Peng, Gao Wang, Jun Ma, Man Chong Leong, Chris Wakefield, James Melott, Yulun Chiu, Di Du, and John N Weinstein. 2018. SoS Notebook: an interactive multi-language data analysis environment. *Bioinformatics* 34, 21 (05 2018), 3768–3770. https://doi.org/10.1093/bioinformatics/bty405

[44] Gordon D. Plotkin. 2004. A Structural Approach to Operational Semantics. *Journal of Logic and Algebraic Programming* 60–61 (2004), 17–139. https://doi.org/10.1016/j.jlap.2004.05.001 Reprint of Technical Report FN-19, DAIMI, Aarhus University, 1981.

[45] David Rauch, Patrick Rein, Stefan Ramson, Jens Lincke, and Robert Hirschfeld. 2019. Babylonian-style Programming: Design and Implementation of an Integration of Live Examples into General-purpose Source Code. *The Art, Science, and Engineering of Programming* 3, 3 (feb 2019). https://doi.org/10.22152/programming-journal.org/2019/3/9

[46] Patrick Rein, Jens Lincke, Stefan Ramson, Toni Mattis, and Robert Hirschfeld. 2017. Living in Your Programming Environment: Towards an Environment for Exploratory Adaptations of Productivity Tools. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Programming Experience (PX/17.2)*. ACM, New York, NY, USA, 17–27. https://doi.org/10.1145/3167108

[47] Patrick Rein, Stefan Ramson, Jens Lincke, Robert Hirschfeld, and Tobias Pape. 2018. Exploratory and Live, Programming and Coding. *The Art, Science, and Engineering of Programming* 3, 1 (Jul 2018). https://doi.org/10.22152/programming-journal.org/2019/3/1

[48] Daniel J. Rough and Aaron Quigley. 2020. End-User Development of Experience Sampling Smartphone Apps -Recommendations and Requirements. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 4, 2 (June 2020), 1–19. https://doi.org/10.1145/3397307

[49] Chanchal Roy and James Cordy. 2007. A Survey on Software Clone Detection Research. *School of Computing TR 2007-541* (01 2007).

[50] Adam Rule, Aurélien Tabard, and James D. Hollan. 2018. *Exploration and Explanation in Computational Notebooks*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/3173574.3173606

[51] D. W. Sandberg. 1988. Smalltalk and Exploratory Programming. *ACM SIGPLAN Notices* 23, 10 (Oct. 1988), 85–92. https://doi.org/10.1145/51607.51614

[52] Beau Sheil. 1986. DATAMATION®: POWER TOOLS FOR PROGRAMMERS. In *Readings in Artificial Intelligence and Software Engineering*, Charles Rich and Richard C. Waters (Eds.). Morgan Kaufmann, 573–580. https://doi.org/10.1016/B978-0-934613-12-5.50048-3

[53] Jeremy Smith and Jonathan Indig. 2020. Polynote. https://polynote.org. (2020). https://polynote.org [Online, accessed 16 July 2021].

[54] Krishna Subramanian, Ilya Zubarev, Simon Völker, and Jan Borchers. 2019. Supporting Data Workers To Perform Exploratory Programming. In *Extended Abstracts of the 2019 CHI Conference on Human Factors in Computing Systems (CHI EA '19)*. ACM, 1–6. https://doi.org/10.1145/3290607.3313027

[55] Marcel Taeumel, Patrick Rein, and Robert Hirschfeld. 2021. *Toward Patterns of Exploratory Programming Practice*. Springer International Publishing, Cham, 127–150. https://doi.org/10.1007/978-3-030-76324-4_7

[56] J. Trenouth. 1991. A Survey of Exploratory Software Development. *Comput. J.* 34, 2 (01 1991), 153–163. https://doi.org/10.1093/comjnl/34.2.153

[57] L. Thomas van Binsbergen, Lu-Chi Liu, Robert van Doesburg, and Tom van Engers. 2020. eFLINT: A Domain-Specific Language for Executable Norm Specifications. In *Proceedings of the 19th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE 2020)*. ACM. https://doi.org/10.1145/3425898.3426958

[58] L. Thomas van Binsbergen, Peter D. Mosses, and Neil Sculthorpe. 2019. Executable Component-Based Semantics. *Journal of Logical and Algebraic Methods in Programming* 103 (feb 2019), 184–212. https://doi.org/10.1016/j.jlamp.2018.12.004

[59] L. Thomas van Binsbergen and Neil Sculthorpe. 2018. The Haskell Funcon Framework. Hackage. (2018). https://hackage.haskell.org/package/funcons-tools [Online, accessed 5 August 2022].

[60] L. Thomas van Binsbergen, Mauricio Verano Merino, Pierre Jeanjean, Tijs van der Storm, Benoit Combemale, and Olivier Barais. 2020. A Principled Approach to REPL Interpreters. In *Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2020)*. ACM, 84–100. https://doi.org/10.1145/3426428.3426917

[61] Tijs van der Storm. 2013. Semantic deltas for live DSL environments. In *2013 1st International Workshop on Live Programming (LIVE)*. 35–38. https://doi.org/10.1109/LIVE.2013.6617347

[62] Mauricio Verano Merino, Jurgen Vinju, and Tijs van der Storm. 2020. Bacatá: Notebooks for DSLs, Almost for Free. *The Art, Science, and Engineering of Programming* 4, 3 (Feb 2020). https://doi.org/10.22152/programming-journal.org/2020/4/11

[63] Bret Victor. 2011. ScrubbingCalculator. http://worrydream.com/ScrubbingCalculator/. (2011). http://worrydream.com/ScrubbingCalculator/ [Online, accessed 16 July 2021].

[64] Michał Walicki and Sigurd Meldal. 1997. Algebraic Approaches to Nondeterminism – an Overview. *Comput. Surveys* 29, 1 (March 1997), 30 – 81. https://doi.org/10.1145/248621.248623

[65] Nathaniel Weinman, Steven M. Drucker, Titus Barik, and Robert DeLine. 2021. Fork It: Supporting Stateful Alternatives in Computational Notebooks. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems (CHI '21)*. ACM, New York, NY, USA. https://doi.org/10.1145/3411764.3445527

[66] Yifan Wu, Joseph M. Hellerstein, and Arvind Satyanarayan. 2020. B2: Bridging Code and Interactive Visualization in Computational Notebooks. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology (UIST '20)*. ACM, New York, NY, USA, 152–165. https://doi.org/10.1145/3379337.3415851