# Computational Notebooks on top of an Exploring Interpreter

**Joey Lai**
Joey.Lai@student.uva.nl

August 24, 2021, 60 pages

**Academic supervisor:** Dr. L.T. (Thomas) van Binsbergen, l.t.vanbinsbergen@uva.nl
**Daily supervisor:** Dr. L.T. (Thomas) van Binsbergen, l.t.vanbinsbergen@uva.nl
**Host organisation/Research group:** Complex Cyber Infrastructure (CCI), https://cci-research.nl/

Universiteit van Amsterdam
Faculteit der Natuurwetenschappen, Wiskunde en Informatica
Master Software Engineering
http://www.software-engineering-amsterdam.nl

# Abstract

Computational notebooks that combine code, documentation and results are very popular among data scientists to do exploratory programming tasks. However, current computational notebooks suffer from a lack of support for experimentation and recording and sensemaking of the exploration history in order to fully support exploratory programming.

In this thesis, we have developed and implemented a computational notebook on top of an exploring interpreter to tackle the problems of current computational notebooks. The potential of using an exploring interpreter for the back-end of computational notebooks has been shown in several experiments with a usability study among a group of participants.

Most participants argued that our computational notebook is able to support exploratory programming better than current computational notebooks in the domain of experimentation and recording and sensemaking of the exploration history.

The results highlight that an exploring interpreter may be a solution to various problems of computational notebooks when used for exploratory programming by developing notebooks on top of an exploring interpreter or integrating the exploring interpreter within current notebooks.

# Contents

# Chapter 1

# Introduction

Currently, exploratory programming is used to understand and support creative programming tasks [1]. According to Kery and Myers, exploratory programming is defined as a task with two different properties [1]. Firstly, code is written as a means to prototype or experiment with different ideas. Secondly, the goal of a programmer changes during the course of writing code. These two properties make exploratory programming suitable for creative programming tasks or when a domain is not that known yet. Exploratory programming is relevant and needed in a variety of applications such as data science [2, 3] and Software Engineering [4, 5].

Computational notebooks are one example of a tool that is used to support exploratory programming. A computational notebook is a read-eval-print-loop (REPL) based environment that is highly interactive and very popular among data scientists [6, 7]. However, traditional computational notebooks are limited in their support for exploratory programming [8–12]. This outcome is mainly due to the fact that it is hard to maintain or recover an earlier approach [1, 8, 13]. Furthermore, these notebooks also lack support for recording and sensemaking of exploration history to compare past attempts or understand past code snippets [1, 8].

Traditional notebooks are mostly used to serve as a collection of code snippets and are not designed to keep a historical execution log that can keep track of the different approaches of notebook users [13]. In order to reproduce past approaches or recall effects of code cells, a user needs to put in a lot of effort to organize and re-execute cells [8, 14].

These issues are important because programmers need a variety of tools that can help them to experiment easily and help them understand a past decision and the effect of code snippets for exploratory programming [1].

In this thesis, we will explore a novel approach based on so-called exploring interpreters [15] for building REPL back-ends to tackle these issues. In computational notebooks, an exploring interpreter can serve as a generic algorithm for executing programs to record resulting runtime states in an execution graph. Essentially, the exploring interpreter is a bookkeeping device that is suggested to be able to provide many desirable features for exploratory programming in computational notebooks that are not yet present.

## 1.1 Research questions

The goal of this work is to evaluate the extent to which it is practical to use an exploring interpreter for the back-ends of computational notebooks to better support exploratory programming. In our research, we will try to tackle the problem of a lack of support for recording and sensemaking of exploration history for exploratory programming and experimentation in computational notebooks by using an exploring interpreter. Therefore, for our thesis, we will need to answer the following research questions:

- RQ1) How practical are exploring interpreters for building computational notebooks to support exploratory programming tasks?
- RQ2) In what ways can we support exploratory programming in computational notebooks on top of an exploring interpreter?
  - RQ2a) To what extent does an 'exploring interpreter' solve the lack of support for experimentation in computational notebooks in the domain of exploratory programming?
  - RQ2b) To what extent does an 'exploring interpreter' solve the lack of support for record-

ing and sensemaking of exploration history for computational notebooks in the domain of exploratory programming?

To answer our research questions, we conduct a usability study with JupyterLab (in Python3) and a computational notebook on top of an exploring interpreter. For the usability study, our computational notebook will be implemented in the eFLINT language (a domain-specific language for formalizing norms) and discuss how our notebook performs against JupyterLab to support exploratory programming.

Firstly, practicality will be evaluated in how usable both notebooks are for completing exploratory programming tasks. Secondly, experimentation will be assessed in how easy it is for users to create, manage and recover past approaches or ideas. Lastly, recording and sensemaking will be measured on how difficult it is for users to recall past effects of code and compare the different versions of the computational notebook.

## 1.2 Contributions

This thesis provides a generic design and an implementation of computational notebooks built on top of an exploring interpreter. The design is language independent such that it is possible to implement the notebook on top of an exploring interpreter for every sequential language.

In particularly, our research makes the following contributions:

1. A generic design of several features that support exploratory programming by taking advantage of an exploring interpreter (chapter 3).
2. Implementation of several features as part of a notebook to enable users to perform experiments (chapter 4).
3. A demonstration of the generic design by implementing the notebook on the Idris language with a large degree of reuse from the eFLINT implementation (chapter 4).
4. An evaluation of the design by running experiments with potential users as testers using an eFLINT notebook (chapter 5).

## 1.3 Outline

In Chapter 2, we describe the background of this thesis. Chapter 3 and Chapter 4 respectively describe the design decisions and the implementation of the computational notebook on top of an exploring interpreter. Evaluation is done in Chapter 5, while limitations and future work are discussed in Chapter 6. Chapter 7 contains the work related to this thesis. Finally, we present our concluding remarks in Chapter 8.

# Chapter 2

# Background

This chapter will present the necessary background information for this thesis. First, we define some basic terminology that will be used throughout this thesis. Next, we will delve into the existing problems of computational notebooks for exploratory programming.

## 2.1 Exploratory programming

Exploratory programming is the process of writing code by trial and error and is considered to be an important process of the software engineering cycle to try some designs to see what works in a new domain or when a requirement is not fully specified before implementation [1, 16]. This development style is a practice that makes it easy and fast to interactively prototype or debug small code applications compared to the standard edit-compile-run-debug practice. In this thesis, we will focus on two different characteristics that define and enable exploratory programming tasks provided by Kery and Myers [1].

### 2.1.1 Recording and sensemaking of the exploration history

In exploratory programming, programmers make use of the history of code as a common way to record their experimentation [1]. A recording of the exploration history that records code and notebook artifacts (such as past variable values or versions of output) is argued to be useful to answer questions about the effect of code snippets during exploratory programming [17]. Therefore, Kery and Myers state that "it is important that a programmer has a way to record specific artifacts (recording) to help them understand (sensemaking) a past decision and its effect" [1]. Support for recording and sensemaking of the exploration history (not only code) can help programmers to reproduce an exploration or compare explorations with past explorations to decide what ideas they would want to use for further explorations.

### 2.1.2 Ease of experimentation

According to Kery and Myers, there are several factors in the domain of exploratory programming that influences the rapid prototyping of programmers [1] during an exploration. Rapid prototyping is described as an activity to experiment and explore different design possibilities instead of continuously building on a fixed requirement. In their study, Kery and Myers identify that viscosity is an important factor that determines the ease of experimentation [1], which in turn affects the process of rapid prototyping.

In exploratory programming, viscosity is defined as a measure of how easy or difficult it is to change code after it is created [18] while preserving the original design. A tool or environment that has a high viscosity will make it harder to make changes while keeping the design of the program. On the other hand, if a low viscosity is achieved, it is easy to maintain the original design and add new code. Kery and Myers state that "viscosity must take into account how easy it is both to create a change and to revert it" [1].

Kery and Myers argue that programmers must be able to easily maintain or recover earlier explorations in order to enable rapid prototyping during exploratory programming. In the paper, the authors claim that this can lower the risks of making exploratory edits in exploratory programming environments. In the context of this thesis, explorations are alternatives produced by executing different code snippets in various order. Studies have found out that a simple undo functionality in exploratory programming

is not enough for programmers because sometimes they would like to invoke undo commands multiple times [5] or continue a previous exploration [1].

In conclusion, the authors state there is a need for clear ways to revert to previous versions or explorations in exploratory programming tools in order to lower the viscosity such that people are able to experiment without much effort [1].

## 2.2 Read–eval–print loop

A read–eval–print loop (REPL) is an interactive computer programming environment that facilitates exploratory programming better than the classic edit–compile–run–debug cycle. Generally, the programming environment takes a single program fragment as input from the user at a time and returns the output to the programmer after executing them. Afterward, it will wait for the next input. This activity can be seen as a dialogue with your application to inquiry about its current configuration. Therefore, a REPL gives users immediate feedback in response to small snippets of code, which makes REPL suitable for lightweight testing, experimentation and debugging. It contains the following three functionality/states that are carried out in order:

- read function: the REPL is able to read or accept a single program fragment from a programmer.
- eval function: the REPL is able to evaluate the given expression or input from the read function.
- print function: the REPL is able to take the result from the eval function and display it to the programmer.

Essentially, a REPL creates a loop with the repeated composition of the read, eval and print function (in that order). REPL environments promote a programming style of typing and executing code step by step in order to check the output of each. This style is also known as incremental programming.

## 2.3 Computational notebooks

According to Rule *et al.*, computational notebooks are single documents that make use of a combination of code, visualizations, and text to construct a computational narrative [19]. A narrative is described as a sequence of events that are structured in order and connected with each other. This sequence of ordered and connected events is stated as an important feature to enable people to document and share their work and results. Notebooks have seen an increase in usage, owing to various free/open-source notebooks, such as MATLAB [20], Jupyter notebook [6], R studio [21], Apache Spark Notebook [22], and various others. These types of notebooks make use of cell blocks that can contain either code or rich text. A programmer can execute or run these cells to create documentations, generate output, or show visualizations. Rule *et al.* mentions that "the cells can be reshuffled to any place or executed in any desired sequence even if these cells are linearly displayed" [19]. Furthermore, Wolfram emphasizes the following: "The idea of a notebook is to have an interactive document that freely mixes code, results, graphics, text and everything else" [23].

Computational notebooks can be seen as a REPL-like tool or environment. In the development environment, the computational notebook can take one or multiple cells as single user input from the programmer. The computational notebook executes these inputs and returns the result to the user afterward. After returning the results, the notebook will return to its read state, which can be seen as a loop. The loop can only be terminated when you close the notebook. Therefore, a computational notebook is a tool that can enable exploratory programming.

## 2.4 Exploratory programming problems in computation notebooks

### 2.4.1 Lack of support for recording and sensemaking of the exploration history

Most computational notebooks lack a lot of support for recording and sensemaking of the exploration history. Although the fact that code cells in a computational notebook are linearly displayed, a programmer can execute them out of order. This can create a messy history of the execution trace that does not match the current state of the notebook. This situation is described by Chattopadhyay *et al.* as a

form of history-associated pain points present in notebooks [8]. For example, messiness in the notebook can increase during an ongoing exploration which can make programmers lose track of their thought processes. The author argues that users need to painfully debug and track how a notebook enters a certain state to fully reproduce its result. This statement is also supported by other academic studies [19, 24] and industry articles [25] for notebook programmers.

Provenance work in computational notebooks is described as a way to track how a specific state or output can be attained to enable reproduction and recall effects of operations [26]. In a study about the uses of computational notebooks for data analysts during an exploration, Rule *et al.* found out that "provenance was useful for keeping track of what analyses had been tried, even if they led to dead ends, keeping older versions of figures in case an advisor decided they preferred them to the new one, and helping analysts untangle exactly how they achieved a result. While analysts can use computational notebooks to track their every step, it does not happen automatically, especially when cells are overwritten and re-run" [19].

R Studio and Jupyter notebooks provide a way to manually display the input history from the current session using specific commands. An example of Jupyter notebook %history command is illustrated in Figure 2.1. However, this list of inputs only contains the raw code lines that have been executed. A list of only code lines lacks any context on the output, effect and thought process of the executed code.

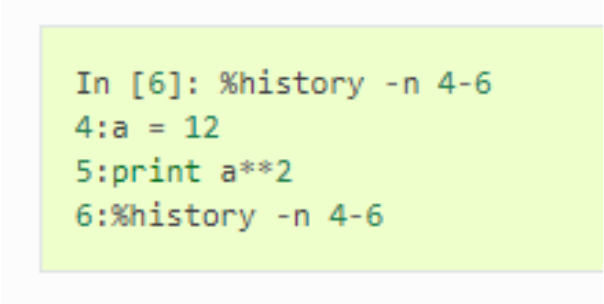



**Figure 2.1: %history command of Jupyter notebook for displaying the input history.**

Additionally, computational notebooks do not have a way to record the internal state of past notebook versions. Most computational notebooks such as RStudio only contain the current internal state of the notebook [21], or they only record code, documentation and output from previous states using various versioning tools. Chattopadhyay *et al.* also considered this as one of the many pain points of computational notebooks. The authors state that "preserving the history of changes and states within and between notebooks is unsupported, leading to unnecessary rework" [8]. In their study, they found out that normal version control systems do not always help programmers successful run a notebook because they do not track the history and execution order of computational notebooks.

### 2.4.2 Lack of support for experimentation

In subsection 2.1.2, we mentioned that viscosity is determined by how easy it is to make changes to programs and revert them. In current computational notebooks, programmers can easily edit and rerun some program snippets in code cells while changing their original design or purpose. However, users of computational notebooks have been seen to struggle a lot when they want to revert changes that they have made during the course of exploration [8]. Therefore, computational notebooks are a relatively high viscous system where programmers could be discouraged to do exploratory programming tasks. A computational notebook with a high viscosity makes it hard to maintain and recover previous explorations that a programmer has explored.

Lack of clear functionalities to revert to a previous state or undo previous code snippets are one of the main reasons why it is so hard to maintain and recover past explorations. In traditional computational notebooks, it is hard or impossible to undo a given executed code cell to revert to a previous state [13]. This can make it difficult to experiment with different variations of a code cell during the course of exploration tasks. In order to experiment with different alternatives of a code cell when variables or configuration are changed in each iteration, a programmer needs to either rerun all the previous code in the exact order from the start or execute a different code to return to the previous state.

Jupyter Notebooks have a feature that enables programmers to create checkpoints that they can use to save their progress and revert back to it. However, this feature is a simplistic kind of recovering past

explorations. Jupyter checkpoints can only be used to save the latest version that you marked. This behaviour means that once you revert back to the checkpoint, you can not revert to an older checkpoint.

## 2.5 Sequential languages

Van Binsbergen et al identify the class of languages that are accepted by REPL interfaces as 'sequential languages' [15]. The authors propose that the class of sequential languages is defined as the class whose languages have the property that a sequence of valid programs is itself a valid program and that the behavior of executing the sequence as one is equivalent to executing its elements one-by-one.

The effect of a program can be represented as a transition function that expresses transitions between two configurations. A configuration contains all necessary contextual information needed to determine the behavior of subsequent programs. Therefore, a set of programs can be combined into a new program where the resulting effect is the same as the sequential effects of each individual program.

In [15], the authors make clear that "sequential languages are able to model software languages with deterministic properties and non-deterministic properties when those properties can be algebraically captured".

## 2.6 eFLINT

eFLINT is a domain-specific language (DSL) for formalizing norms. The theoretical bases of the DSL are created using Hohfeld's framework of legal fundamental conception that is able to formalize norms from various sources. eFLINT is used to formalize norms that are found in laws, regulations, policies, contracts and (data-sharing) agreements [27].

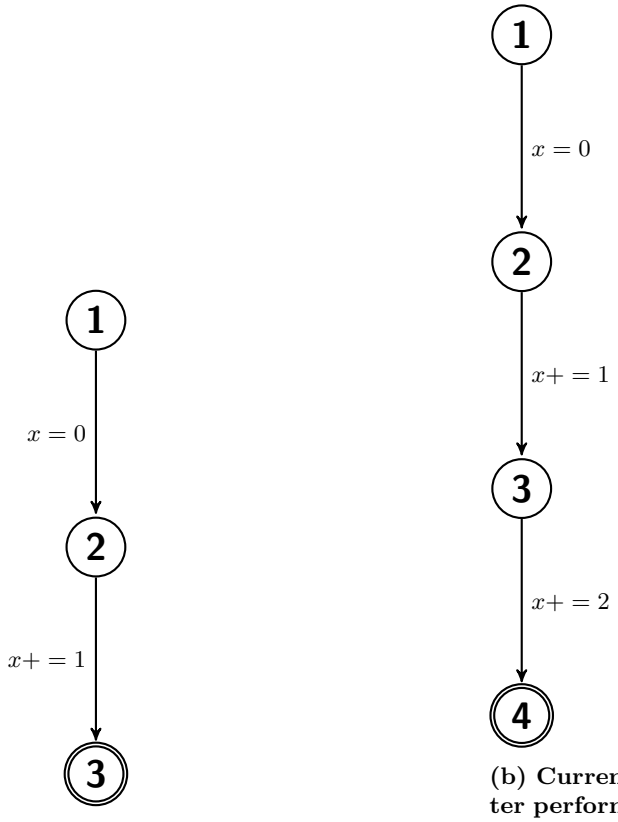
## 2.7 Exploring interpreters

Exploring interpreters can be used for sequential languages to promote certain design principles. The core principles for building a REPL for some base language on top of an exploring interpreter for a sequential language are stated in [15] as follows:

- the effects of a code snippet manifest as changes to an explicit state representation (a configuration)
- the effects of a code snippet are determined by the definitional interpreter used by the exploring interpreter
- the effects of a sequence of code snippets is the composition of the effects of the individual snippets
- only code snippets change configurations

An exploring interpreter can serve as an algorithm layer on top of a definitional interpreter for sequential languages that records a directed execution graph of the programming environment. According to Van Binsbergen *et al.*, "a definitional interpreter of the language determines the effects of individual code snippets as well as the effect of their compositions" [15]. In the execution graph resulted from an exploring interpreter, the nodes are represented by the different configuration or states of the REPL environment, while the edges are the code snippets that creates the transition from one state to another. Therefore, a source and target node of a transition are respectively the previous state and next state in the programming environment. Each node in the graph is also called a configuration. The leaf nodes in this execution graph are nodes that do not have a path or transition to a different node and represents the end of all the explorations that a programmer has explored; in other words, they do not have a target node.

In any state of a REPL environment, the exploring interpreter can perform an 'execute' or 'revert' action. The 'execute' command creates a transition from the current state of the environment to a new state. An 'execute' command takes a phrase as input. After this command, the execution graph gets updated with a new node and its corresponding edge. On the other hand, a 'revert' action changes the current state of the environment to an input state selected by the user. An example of the 'execute' action is shown in Figure 2.2.




**(a) Current runtime state is node 3.**

**(b) Current runtime state is node 4 after performing an 'execute' action with 'x+=2' (phrase input) as code snippet.**

**Figure 2.2: Illustration of performing an 'execute' action on the current state of the programming environment with an exploring interpreter.**

An exploring interpreter can have a destructive property or a sharing property. When the destructive property is enabled, the exploring interpreter will remove the successive children nodes after reverting to a specific node. These configurations result in an execution graph that is maintained as a single path. When sharing is enabled, the exploring interpreter will only create new nodes and edges in the execution graph after an 'execution' action if the resulting configuration did not exist yet.

In this thesis, we propose a design of computational notebooks that are built on top of an exploring interpreter to solve the problems of computational notebooks in the domain of exploratory programming. The exploring interpreter is an algorithm that should be possible to implement in every language of the back-end kernel of computational notebooks due to the sequential nature of incremental programming languages in REPL environments.

An exploring interpreter can provide us with an execution graph with configurations that serves as a way to collect provenance work automatically that not only records code but also runtime states. In this graph, the nodes or configurations are represented by the different runtime states or versions of the notebook, while the edges are the code snippets that creates the transition from one notebook state to another. Each edge has a source and target identifier that respectively corresponds with the source node (previous notebook version) and target node (resulting notebook version). Executing a single code program in this computational notebook will update this graph with an additional node and edge as seen in Figure 2.2. A code snippet of multiple singular programs will result in multiple edges and nodes by the exploring interpreter.

Furthermore, the 'revert' functionality of the exploring interpreter can give programmers the ability to quickly make changes and revert them to lower the viscosity and therefore reduce the risk of making exploratory edits.

# Chapter 3

# Design

The contents of this chapter has been incorporated into a paper that is under submission:

*(Under submission).* A Language Independent Protocol for Exploratory Programming. Mauricio Verano Merino, L. Thomas van Binsbergen, Pierre Jeanjean, Damian Frolich, Joey Lai, Tijs van der Storm, Benoit Combemale, and Olivier Barais. Proceedings of the 14th ACM SIGPLAN International Conference on Software Language Engineering (SLE 2021). ACM.

The goal of this thesis is to demonstrate the potential of computational notebooks on top of an exploring interpreter in the domain of exploring programming. We want to show how exploring interpreters support the basic and fundamental functionalities/requirements of computational notebooks for exploratory programming while also tackling the issues that have been stated in the introduction. In this chapter, we will present and discuss our design decisions of several additional functionalities that we have implemented in computational notebooks using an exploring interpreter to better support exploratory programming.

## 3.1 Components layout

Our computational notebook on top of an exploring interpreter is made up out of 3 main components arranged as columns next to each other, and an example of our interface is shown in Figure 3.1. Each of these components can be hidden in order to focus on a specific view. In its normal view, the components are positioned is as follows:

- The left column contains the traditional computational notebook interface where a user is able to create, delete or interact with cells. This left column is made out of code and documentation cells as provided in most computational notebooks. Each cell has an 'action' button where a programmer is able to perform an 'execute' or 'revert' action with the exploring interpreter that respectively executes a code cell or reverts the computational notebook to a previous notebook state. Furthermore, a user can use the action button in each cell to switch between different versions of the code cell.
- The middle column contains the execution trace that represents the current runtime state and of the programming environment and the states preceding it within its execution trace. The execution trace contains buttons for each node where a user is able to annotate or tag a specific node. Furthermore, a user can also undo several executed code snippets in the current trace by reverting to a previous node in the execution trace or convert the node to a new code cell in the left column.
- The right column contains tabs that automatically or manually keep track of the different versions of the notebook. The head nodes are tracked automatically by the notebook whenever a user successfully executes a code snippet. On the other hand, the tracked nodes get updated whenever a user tags a node in the execution trace. A node can be removed from either list by pressing the 'remove node' button.

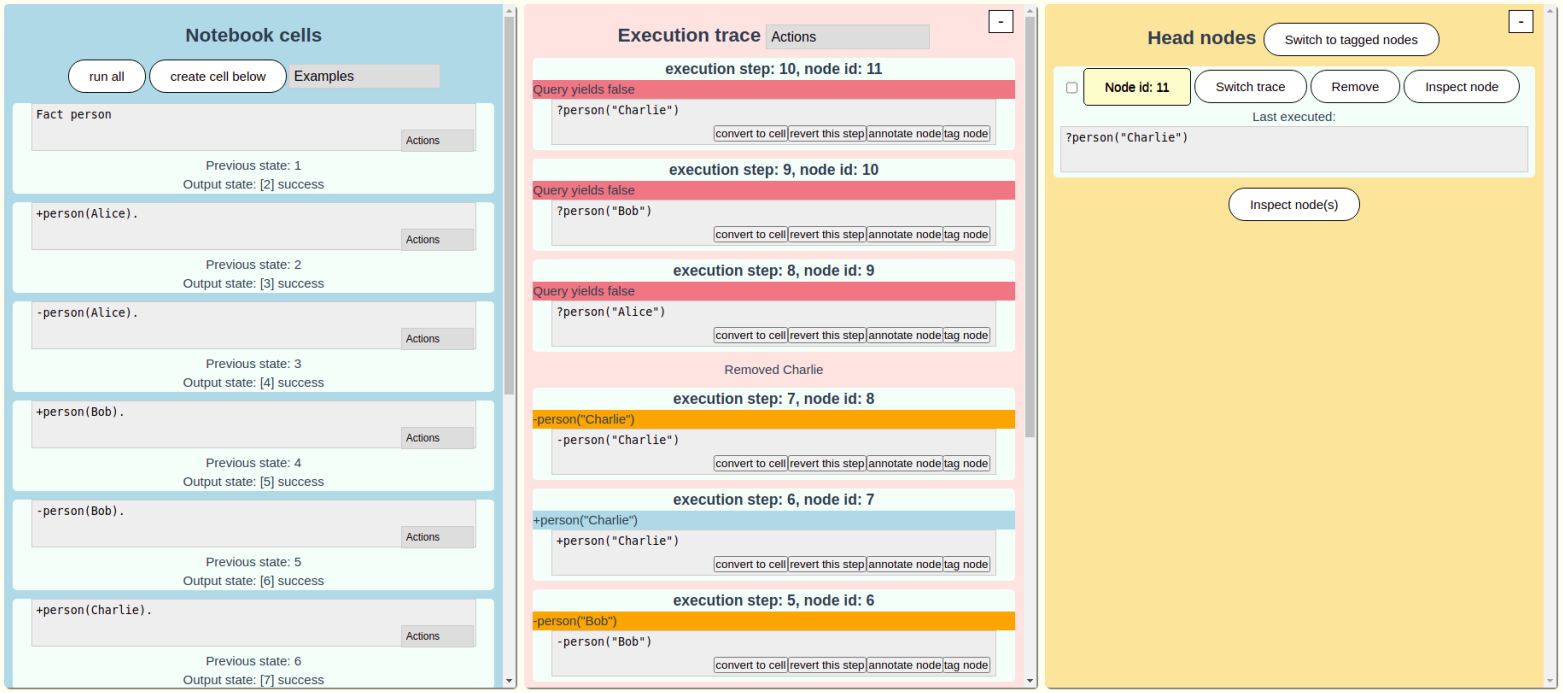



**Figure 3.1: Layout of the computational notebook on top of an exploring interpreter with the eFLINT language.**

## 3.2 Sharing and destructive properties of the exploring interpreter

As mentioned in section 2.7, an exploring interpreter can enable sharing and destructive properties. However, we argue that a computational notebook on top of an exploring interpreter should not have these properties for exploratory programming.

When sharing is disabled, the notebook will produce an execution graph where each node or runtime state of the notebook only has one execution path. This can remove ambiguity and confusion when there are multiple paths to the same runtime state or node in the computational notebook for the programmer when sharing is enabled.

When the destructive property is enabled, the computational notebook will remove nodes or edges from the execution graph whenever a user reverts to a previous runtime state of the programming environment. However, this is not favorable for exploratory programming because the recording of the exploration history is important, as explained in section 2.1. During exploratory programming in computational notebooks, we do not want to remove past approaches or experiments with different combinations of code that have been tried out. Programmers need to be able to look back at past states of the notebooks in order to understand the effects of different approaches in the course of exploring different design ideas. Therefore, a computational notebook on an exploring interpreter should not have this property when it is used for exploratory programming.

The resulting execution graph is a tree where the root node is equal to the initial runtime state of the programming environment. Each leaf node of the graph represents the end of each exploration that a user has explored. An example of this execution graph is illustrated in Figure 3.2.

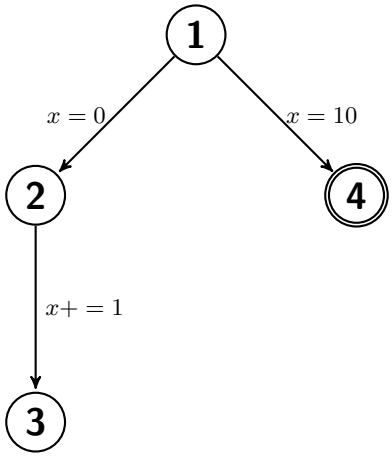



**Figure 3.2: The execution graph of an exploring interpreter (with no sharing property and no destructive property) illustrated with simple math operations. Node 1 is the root node, which represents the initial state of the computational notebook. Node 3 indicates a previous exploration that a user has explored. Node 4 represents the current state of the notebook where the current x value is equal to 10.**

## 3.3 Execution trace

As explained in subsection 2.4.1, there is a lack of support for recording and sensemaking of the exploration history. Because of the out-of-order execution functionality that computational notebooks provide, the internal state of the notebook can quickly become different from notebooks cells that are visible for the programmer. Programmers need history support of code in computational notebooks for exploratory programming to easily debug how a computational notebook has entered a specific state. Furthermore, the exploratory programming tools need to provide a way to record artifacts such that programmers are able to understand the effects of code snippets in order to reason about their past decisions, as stated in subsection 2.1.1.

Therefore, in our computational notebook, we aim to gather more context about the executed code snippets in the exploration history of the notebook states by building a notebook on top of an exploring interpreter. The resulting computational notebook contains the whole execution graph of the current environment that is maintained by the exploring interpreter. Each node (runtime state of the notebook) in the execution graph contains a specific configuration after executing a single program. These configurations contain all necessary information that makes it possible to show the output and effect of the executed code on the state of the notebook.

In our computational notebook, we constantly display the execution trace, the path between the root node and the node of the current runtime state, located in the middle column of our notebook (shown in Figure 3.3). In this execution trace, a programmer has an overview of all the code snippets, with their resulting effect on the state of the notebook that he has executed from the clean state of the notebook to the current state of the programming environment. Therefore, our computational notebook provides the user with a way to collect provenance work automatically and displays it in the execution trace. During exploratory programming, a programmer can make use of this execution trace to understand his past decisions and the effect of each individual program that he has executed.

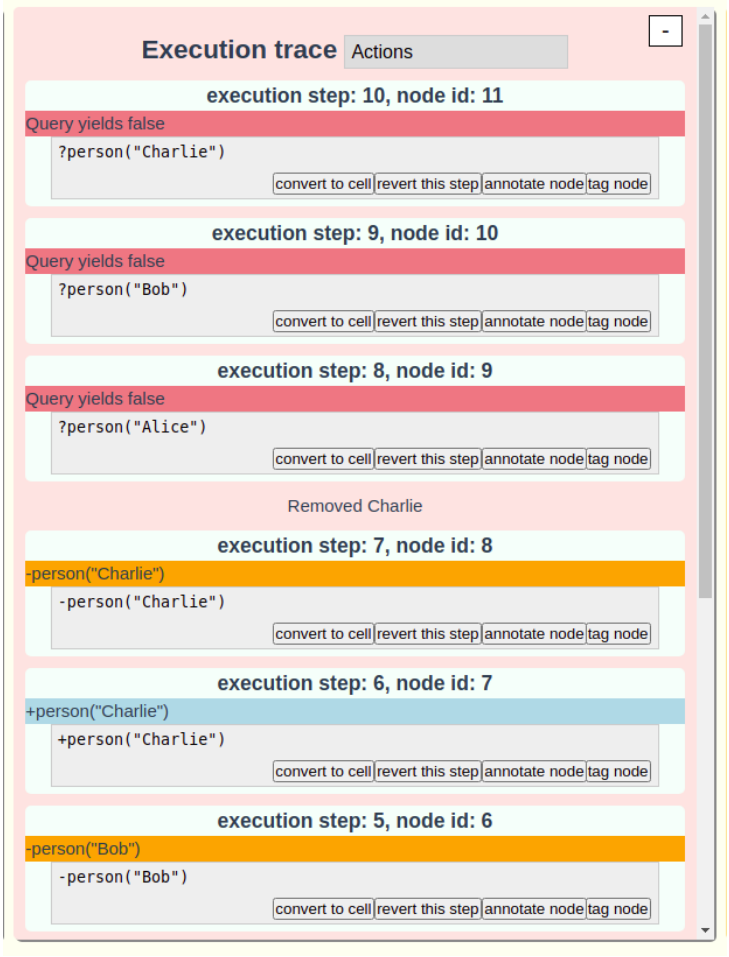



**Figure 3.3: An execution trace of the computational notebook on top of an exploring interpreter with the eFLINT language.**

## Interacting with the execution trace

Users can interact in various ways with the execution trace during exploratory programming. Firstly, a programmer can convert a node in the execution trace to a code cell in the left column of the notebook. The converted cell contains the code snippet that creates the transition from the previous state to the resulting state after execution.

Secondly, each node in the execution trace has a 'revert this step' button that operates as an undo functionality for the notebook to better support experimentation in computational notebooks, which is further explained in section 3.5.

Thirdly, in addition to documentation of code cells, we support a way to create annotations or comments during the provenance work so that programmers are able to document not only code cells but also their thought process or design decisions during an exploration. According to Rule *et al.*, this form of inline journaling can help programmers easily finding the right information across notebooks and improve user experience during an exploration [19]. The annotations or comments are assigned to states of the notebook instead of separate pieces like the documentation cells of current notebooks. In our computational notebook, the annotations are shown above the headers in the execution trace as illustrated above execution step 7 in Figure 3.3.

Finally, we provide users with a 'tag node' button to bookmark nodes of the execution trace in a tagged node list (section 3.4). The tagged node list is located in the right column of the computational notebook, which a programmer can interact with at all times (explained in section 3.5 and section 3.6).

## 3.4 Tracking (the different states of the notebook)

To help the programmer keep track of all their exploration in the computational notebook, we provide an automatic and manual way to maintain, display, and interact with the different states of the notebook located in the right column (shown in Figure 3.4).

Firstly, with automatic tracking, we track the end states of all the explorations that a programmer has explored. This list of notebook states consists of all the leaf nodes of the execution graph. In our computational notebook, we call these leaf nodes the head nodes.

Secondly, with manual tracking, we give programmers the ability to maintain their list of explorations, which can be compared to version control tools where a programmer can record changes on their own. In both lists of tracked nodes, a user can remove a specific node from the list that is not useful anymore.

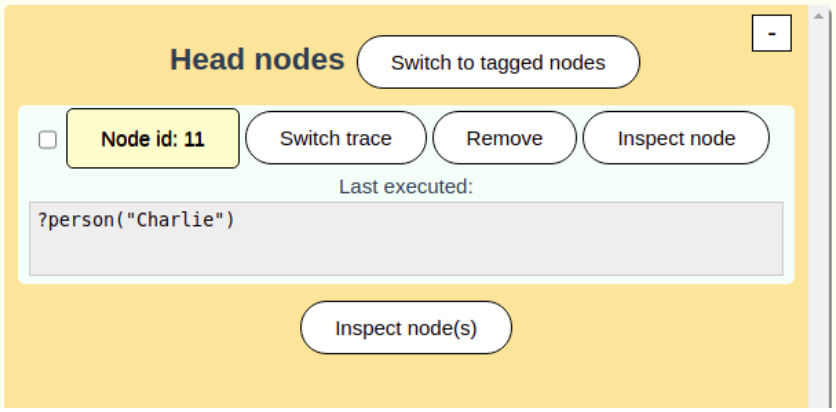



**Figure 3.4: The list of the different explorations (leaf nodes) that a programmer has explored that is tracked automatically.**

## 3.5 Reverting

During exploratory programming in notebooks, a programmer is bound to edit existing code and rerun cells to explore different approaches and design ideas. Therefore, programmers need a way to easily edit and revert changes in order to easily experiment with multiple explorations at the same time to enable exploratory programming. However, as mentioned in section 2.4, most computational notebooks lack a good or clear revert functionality to recover past notebook states or experimental approaches. Current computational notebooks need to rely on traditional versioning tools to provide a way to revert back to a previous state. Yet, traditional versioning tools do not guarantee that rerunning the code versions will help programmers successfully reproduce their original state.

In our computational notebook, we allow programmers to revert to different runtime states of the notebooks to aid them in recovering old experiments and reducing the viscosity of computational notebooks. At any point, a programmer can perform the 'revert' action. The revert functionality of a notebook on top of an exploring interpreter makes it possible for a programmer to revert to a different runtime state of the notebook without the need to rerun previous code or execute a specific code snippet. This functionality provides a clear way for a user to recover old explorations without much effort. Lessening the effort to return previous explorations will make it easier for programmers to experiment with different ideas and thus reduce the risk of making exploratory edits.

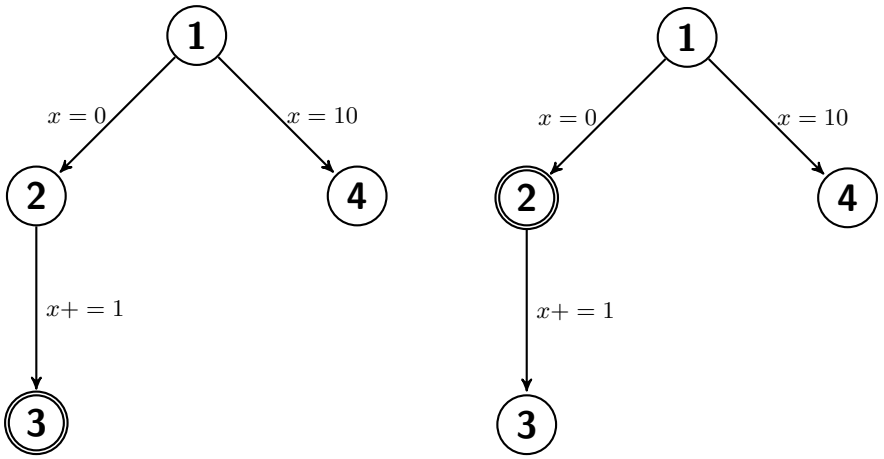
Firstly, we allow users to change the current state of the notebook to the state before a user has executed a specific code cell using the 'revert this cell' under the 'action' button in each code cell. Furthermore, each cell maintains a list of edges from previously executed code snippets that a user can make use of to display and switch between the different versions of code and output. This functionality can address the issue of losing track when rerunning code cells in current computational notebooks.

Secondly, we provided a way for users to revert the programming environment to a previous notebook state in the current execution trace using the 'revert this step' button. The 'revert this step' button operates as an undo functionality to help programmers experiment more easily than the undo functionality in traditional computational notebooks.

Lastly, a user can revert to nodes that are tracked by the notebook or the user by pressing the 'switch trace' button in the right column of the notebook. The tracked nodes are the leaf nodes of the execution

graph or any bookmarked node by the user, as explained in section 3.4. The switch trace button can make it easy for programmers to recover past notebook versions or experimental approaches.

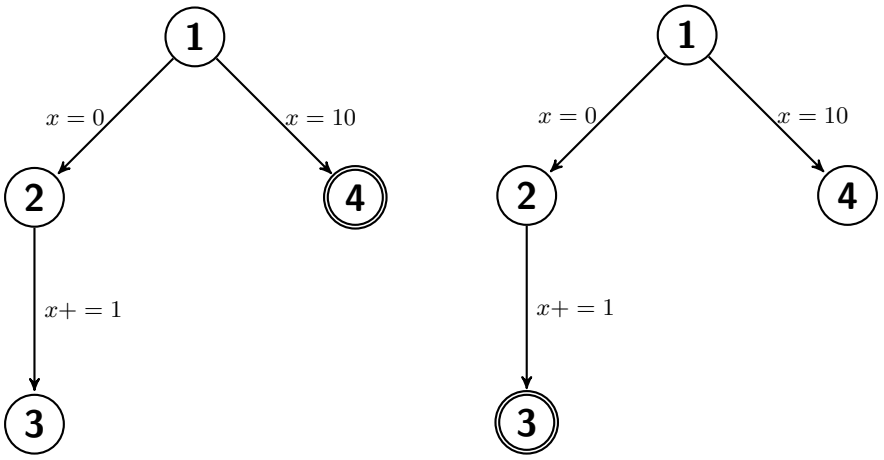
In Figure 3.5 and Figure 3.6, we illustrate how the exploring interpreter reverts the current state of the notebook to a different state with the execution graph using the 'revert this step' and 'switch trace' buttons.




**(a) Current state is node 3 and the execution trace is the path between node 1 and 3 before reverting.**

**(b) Current state is node 2 and the execution trace is the path between node 1 and 2 after reverting to node 2.**

**Figure 3.5: Illustration of reverting to a previous notebook state in the execution trace (revert this step) with an exploring interpreter.**




**(a) Current state is node 4 and the execution trace is the path between node 1 and 4 before reverting.**

**(b) Current state is node 3 and the execution trace is the path between node 1 and 3 after reverting to node 3.**

**Figure 3.6: Illustration of reverting to a head or leaf node (switch trace) with an exploring interpreter.**

## 3.6 Inspecting and comparing the states of the notebook

During the process of exploratory programming, programmers need to be able to compare past approaches with each other or inspect a specific exploration in order to decide what they want to experiment with next [1]. In order to aid programmers to make better decisions when choosing which node they want to revert to and experiment with, we take inspiration from Head *et al.* [9] method, which compares notebook versions by their code, documentation, and output by designing an inspect functionality.

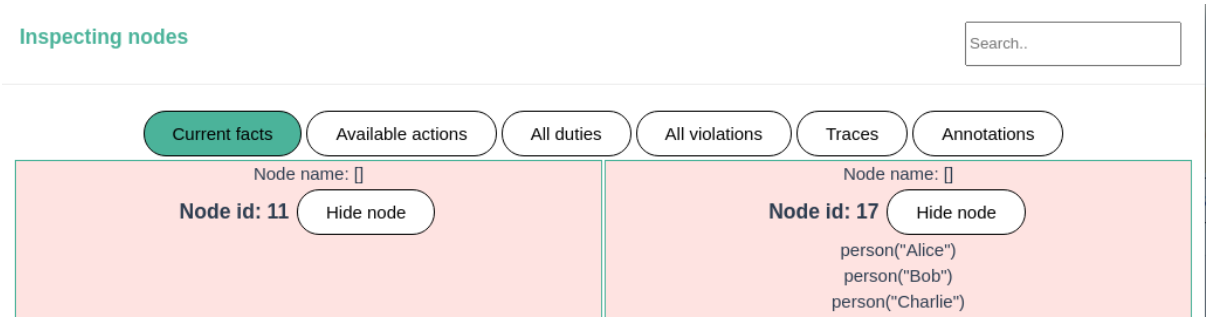
In our inspect functionality ('inspect node(s)' button in the tracked nodes in Figure 3.4), we can

analyze a single runtime state or compare multiple runtime states of the notebook with each other. In this inspection, we can analyze configuration info that is generalized across all sequential languages. For example, we can compare execution traces between runtime states that are associated with each state of the notebook that shows every executed code and their result on the notebook. Other configuration info in runtime states is language-specific such as variable values in functional languages or facts in logical languages.

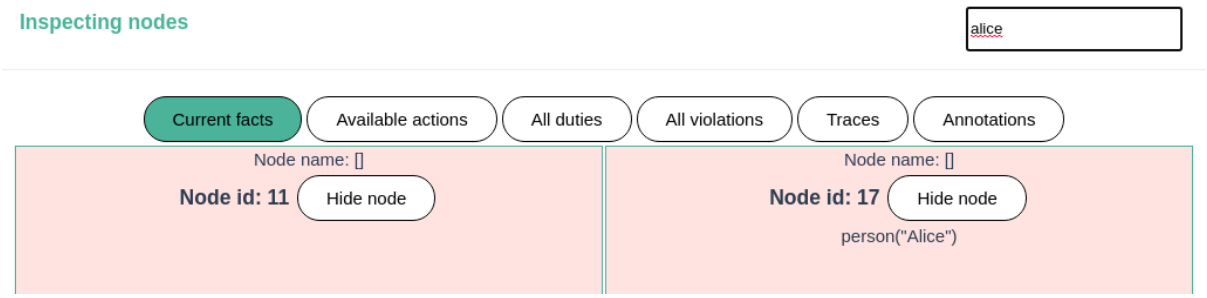
Furthermore, we propose a filter functionality to make it easier to search for information (variables in functional programming or facts in logic programming) that is present in all or some of the different states of the notebook.

Our inspect functionality differs from Head *et al.* [9] in the aspect that we compare configuration information, such as the execution trace and internal runtime state values, instead of the notebook cells and their output. An example of our inspect functionality is shown in Figure 3.7 and Figure 3.8 with the eFLINT language after selecting specific nodes and pressing the 'inspect node(s)' button to create an inspect modal (popup window) view.


**Figure 3.7: Comparing execution traces between different nodes in the eFLINT language. Each execution step can be clicked on to show more information, such as the code snippets that created the nodes.**




(a) No searching/filtering.




(b) Searching/filtering on the string 'alice'.

**Figure 3.8: Comparing different facts between different nodes in the eFLINT language**

Theoretically, it would be possible to apply various techniques to analyze the differences between the states of the notebooks. For example, one could apply string comparison to measure the closeness of the different states by finding the maximum number of identical symbols or produce diffs for the execution traces [28].

## 3.7 Summary

To sum up, we have designed several functionalities that are not present yet in current computational notebooks with the help of an exploring interpreter to support exploratory programming. In order to evaluate our design decisions in the domain of exploratory programming, we will give our implementation of the design using the eFLINT language in the next chapter, which we will use to carry out our experiments.

# Chapter 4

# Implementation

The prototype of the notebook front-end interface is built as a single-page web application implemented in Vue.js. Every interaction with the exploring interpreter in the back-end of our computational notebook is an API request using Axios, a promised-based client for node.js and the browser.

## 4.1 Executing a code snippet

For each code cell, we implemented a standard execution button that executes a code snippet in the cell on the current state of the notebook. This 'execution' action sends a 'phrase' POST request to the back-end with the code snippet as a parameter using the 'executeCell()' function (shown in Listing 4.1). In response, the back-end will send back a response consisting of information about the execution of the code snippet. On successful execution of the code snippet, the back-end will send a promise object as a response containing the effect of a program being the changes it makes to the current configuration, giving rise to a new configuration (as captured by a transition). Each time a code snippet is successfully executed, the notebook will save the transition or edge (to the new state) in the code cell to maintain a list of different code snippets that have been tried out with their effects in the front-end. On errors, the computational notebook will show the errors as messages in alerts.

```
1  // Executing a code cell on the current state of the notebook. The instance
       parameter represents the identifier of the computational notebook
2  executeCell:  function(instance, cellID, codeSnippet) {
3    axiosSendPhrase(instance, codeSnippet)
4      .then((response) => {
5        if (response.errors) {
6          alert(errors);
7        } else {
8          // this.cells represents the list of cells in the left column of the
               notebook
9          this.cells[cellID]['nextState'] = response['new-state']
10         this.cells[cellID]['prevState'] = this.edges.length > 0 ? this.edges[0].
               target_id : 1
11         this.cells[cellID]['output'] = response['output']
12
13         // Maintains a map object of different transitions or edges for each code
               cell after executing a code snippet with the target state as key and
               information (target node, source node, code snippet, effect) as value
14         this.cells[cellID]['versions'][response['new-state']] = {
15           'nextState': response['nextState'],
16           'prevState': this.cells[cellID]['prevState'],
17           'code': codeSnippet,
18           'output': response['output']
19         }
20       }
21       // Updates the execution trace (explained in section 4.2 and Listing 4.2)
22       this.getEdges()
23     })
24 },
25
26 // Creates and executes a command request with a `phrase' as command
27 axiosSendPhrase (instance, codeSnippet) {
28     return axiosSendCommand(instance, {"command" : "phrase", "text": codeSnippet})
29 },
```

```
30
31  // Sends POST requests to the back-end (exploring interpreter) where data contains
        the type command to be performed (execute, revert or requesting information)
32  axiosSendCommand(instance, data) {
33      return axios.post('../server/command', {
34        "uuid" : instance,
35        "request-type" : "command",
36        "data" : data
37      })
38          .then((response) => response.data.response)
39  },
```

**Listing 4.1: Implementation of the 'executeCell()' function using axios and JavaScript for the eFLINT language.**

## 4.2 Execution trace

### 4.2.1 Retrieving an execution trace

The execution trace of the notebook reflects the current state of the programming environment, and it is retrieved by sending a 'history' POST request to the back-end with our 'getEdges()' function (as shown in Listing 4.2). After retrieving this POST request, the back-end will compute and return all edges in the path between the current node and root node. The back-end is easily able to achieve this due to the fact that no sharing is enabled. The disabled sharing property makes sure that all nodes have exactly one source state that represents its previous state in the execution graph. Therefore, there is only one path from the current state of the computational notebook to the clean state that the back-end can compute. Each edge in the path contains the phrase (code snippet), output (effect on the notebook state), source identifier (previous node before execution) and target identifier (resulting node after execution). In order to make sure that the execution trace always corresponds to the current state of the programming environment, we call the 'getEdges()' function after executing a code snippet or reverting to a different state (shown in Listing 4.1 and Listing 4.5).

```
1   // Retrieves the current path of edges of the current notebook state
2   getEdges: function(instance) {
3     axiosGetHistory(instance)
4       .then((response) => {
5           // Updates the execution trace
6           this.edges = response.edges.reverse()
7       })
8   },
9
10  // Creates a command request with a `history' as command
11  axiosGetHistory (instance) {
12      // shown in Listing 4.1
13      return axiosSendCommand(instance, {"command" : "history"})
14  },
```

**Listing 4.2: Implementation of the 'getEdges()' function using axios and JavaScript.**

### 4.2.2 Displaying the execution trace

In our computational notebook, we implemented the execution trace as a stack. Each layer of the stack represents a node (runtime state) of the nodes in the execution path from the initial notebook state to the current notebook state. In each layer, we display the relevant code snippet, annotation and its effect on the corresponding notebook state using the configuration info that is available. The configuration info depends on which language we have as the back-end for the notebook. For instance, we can show mutations of variables in functional languages or additions of facts in logical languages. The layers are arranged in ascending order sorted by last executed. An example of the implementation for the logical language eFLINT is shown in Listing 4.3.

```
1   ...
2   <div v-for="(edge, index) in edges" :key="index">
3       <!-- Display the annotations of the node -->
4       {{annotations[edge.target_id]}}
```

```
5       <!-- Display the execution step in the execution trace and its corresponding
            node identifier -->
6       <h3>execution step: {{edges.length - index}}, node id: {{edge.target_id}}</h3>
7
8       <!-- Output query (success, failure, error) -->
9       <div v-if="edge.output.length != 0">
10          <div v-if="edge.output=='success'" class='successful-query'>
11              "Query yields true"
12          </div>
13          <div v-else-if="edge.output=='failure'" class='unsuccessful-query'>
14              "Query yields false"
15          </div>
16          <div v-else-if="edge.output=='error-type'" class='error-query'>
17              {{edge.output.error-type}}
18          </div>
19      </div>
20
21      <!-- Terminated facts -->
22      <div v-for="(fact, index) in edge.terminated_facts" :key="index">
23          <div class='terminated-facts'>
24              -{{fact.textual}}
25          </div>
26      </div>
27
28      <!-- Created facts -->
29      <div v-for="(fact, index) in edge.created_facts" :key="index">
30          <div class='created-facts'>
31              +{{fact.textual}}
32          </div>
33      </div>
34
35      <!-- Violations -->
36      <div v-for="(violation, index) in edge.violations" :key="index">
37          <div v-if="violation['violation']=='invariant'" class='violated-invariant'>
38              "Violated invariant:" {{violation['invariant']}}
39          </div>
40          <div v-else-if="violation['violation']=='action'" class='action-violated'>
41              "Violated:" {{violation.value.textual}}
42          </div>
43          <div v-else class='tagged-elem duty-violated'>
44              "Violated:" {{violation['value']['textual']}}
45          </div>
46      </div>
47
48      <!-- Code snippet that created this node/notebook state -->
49      <textarea v-model="edge.phrase"></textarea>
50  </div>
51  ...
```

**Listing 4.3: Implementation of converting the edges to a stack in the front-end with Vue.**

### 4.2.3 Documenting the execution trace

Documenting the execution trace is easily done by pressing the 'annotate node' button of the layers in the execution trace. The annotate node button will show a simple textarea in a modal that can be edited by the user. The changes will be automatically updated and shown at the top of each layer. The annotations are maintained as a map object in the front-end, where the node identifier is the key, and the annotation is the value.

### 4.2.4 Converting a node to code cell

A node in the execution trace can be easily converted to a code cell with the 'convert to cell' button. As seen in this section, each node in the execution trace has the code snippet, source id, and target id. The 'convert to cell' button takes these values as parameters to create a new code cell that gets appended to the list of cells.

## 4.3 Tracking (the different states of the notebook)

### 4.3.1 Automatically

As explained in section 3.4, we want to track the end states of all the explorations automatically. In order to do so, we obtain all the head nodes (leaf nodes of the execution graph) with a 'trace-heads' POST request using the getHeadNodes() function (as shown in Listing 4.4). These leaf nodes represent the end states of all different exploration paths that a programmer has explored because each leaf node does not have a transition to a target node. As mentioned in section 3.2, the execution graph gets updated after successfully executing a code snippet in a computational notebook on top of an exploring interpreter. Whenever the execution graph gets updated, there is a chance that the list of head nodes also gets alternated. To make sure that the automatic tracking maintains the correct list of nodes, we retrieve the leaf nodes of the execution graph after every successful execution of a code snippet.

```
1  // Retrieves the current head/leaf nodes of the execution graph
2  getHeadNodes: function(instance) {
3    axiosGetHeads(instance)
4      .then((response) => {
5          this.headNodes = response.nodes.reverse()
6      })
7  },
8
9  // Creates a command request with a `trace-heads' as command
10 axiosGetHeads (instance) {
11     // shown in Listing 4.1
12     return axiosSendCommand(instance, {"command" : "trace-heads"})
13 },
```

**Listing 4.4: Implementation of the 'getHeadNodes()' function using axios and JavaScript.**

### 4.3.2 Manually

In addition to automatic tracking, we want to provide users with a way to track specific nodes manually. A programmer can track or bookmark a specific node in the execution trace by pressing the 'tag' button, which saves the node identifier of the selected node in a list separate from the list generated with the automatic tracking. Our computational notebook also provides a way to create and edit names for each bookmarked node to make it easier for users to search for a specific exploration. The names for the tagged nodes are maintained as a map object in the front-end, where the node identifier is the key, and the name is the value.

### 4.3.3 Displaying the tracked nodes

The nodes or runtime states that have been tracked automatically or manually are displayed in the right column of the computational notebook. Each node contains an identifier and the last code snippet that was executed in order to reach its state.

## 4.4 Reverting

In our computational notebook, we can allow different kinds of revert functionality to aid programmers in their experimentation of explorations tasks and reduce the viscosity. which is stated to be helpful for exploratory programming [1]. As mentioned in section 3.5, the revert functionality of our notebook makes it possible for a programmer to revert to a different state of the notebook without the need to rerun previous code or execute a specific snippet of a code snippet.

Each revert functionality is a revert button that sends a 'revert' POST request to the back-end with the source node identifier using the 'revert' function (as shown in Listing 4.5). After receiving this POST request, the back-end will update its current state to the requested state by the user.

```
1  // Reverts the notebook to the specified notebook state
2  revert: function(instance, sourceID) {
3    axiosRevert(instance, sourceID)
4      .then((response) => {
5        // shown in Listing 4.2
6        this.getEdges()
```

```
7       })
8   },
9
10  // Creates a command request with a `revert' as command
11  axiosRevert(instance, sourceID) {
12      // shown in Listing 4.1
13      return axiosSendCommand(instance, {"command" : "revert", "value": sourceID})
14  }
```

**Listing 4.5: Implementation of the 'revert()' function using axios and JavaScript.**

### 4.4.1 Revert executed code cells

The first revert functionality is implemented in each code cell of the computational notebook under the 'action' button. After executing a code snippet in a code cell, the 'exploring interpreter' will create an additional edge and node in the execution graph. In our computational notebook, we save the identifier of the target node and the source node of the created edge in the code cell that was executed (as explained in section 4.1). A programmer can make use of the source node identifier to revert the notebook state to the state before he has executed the code snippet of the respective cell.

### 4.4.2 Revert to previous states of the execution history

The revert functionality in our notebook can also serve as an undo functionality in the execution trace. The execution trace consists of nodes representing the different states of the notebook with their own unique identification number. A programmer using our notebook is able to select and revert to any past version of the notebook in the execution trace by pressing 'revert this step' button, which will invoke the revert function with the respective node identifier that reverts the notebook state to the state before the execution of the selected node. This functionality enables the user to invoke multiple undo commands in computational notebooks without the need to rerun any code.

### 4.4.3 Revert to nodes that have been tracked by the user/notebook

Alternatively, users can make use of the tracked nodes to revert back to a previous exploration. Every exploration that is tracked automatically by the notebook or manually by the user is located in the right column of the notebook and contains a unique identifier. By pressing the 'switch trace' button, the 'revert' function will be executed with the node identifier of the selected node. In this right column, a user can switch between the two tracked lists by pressing the switch button found at the top of the column tab that toggles a 'true' or 'false' flag variable.

## 4.5 Inspecting and Comparing

### 4.5.1 Creating the modal

Users of the computational notebook on top of an exploring interpreter can select single or several nodes in the head or tagged node list in the right column. By pressing the 'inspect node(s)' button, the computational notebook will bring forth a modal that receives a list of node identifiers generated from the selected nodes. Multiple columns in the inspect modal will be created depending on the number of nodes in the generated list. For each column in the modal, the computational notebook executes a 'getStateEdges()' function that send a request to the back-end using the '*history*' command with a node identifier to retrieve its execution trace (shown in Listing 4.6).

```
1   // Executed for each column in the inspect modal and retrieves the path of edges
        between the root node and the node of the column
2   getStateEdges: function(instance, columnNodeID) {
3     axiosGetStateHistory(instance, columnNodeID)
4       .then((response) => {
5           // Updates the execution trace, each column has its own list of edges.
6           this.columnEdges = response.edges.reverse()
7       })
8   },
9
10  // Creates a command request with a `history' as command
11  axiosGetStateHistory (instance, nodeID) {
```

```
12      // shown in Listing 4.1
13      return axiosSendCommand(instance, {"command" : "history", "state" : state_id})
14  },
```

**Listing 4.6: Implementation of the 'getStateEdges()' function using axios and JavaScript.**

### 4.5.2 Inspecting and comparing

In the inspect modal, programmers can inspect a single runtime state or compare multiple runtime states with each other. Using the available buttons located at the top of the modal, the programmer can choose what kind of information they want to analyze or compare. The buttons make use of a simple switch case using the type of information as the expression to evaluate.

### 4.5.3 Searching and filtering

The filter functionality during the comparison of runtime states is implemented by introducing a search input field in the inspect modal. Filtering is done by a simple text-based search on the document object of the HTML document in the browser. As explained section 3.6, the filtering can be done for different kinds of information. In some functional languages, we can filter on variable names or specific values. On the other hand, in logical languages like Prolog, we can filter on specific facts that are set to true. In Listing 4.7, we show an example of the search functionality implemented on an eFLINT (logical programming language) back-end where we can search for facts, actions, duties or violations. It is a simple but powerful tool to search quickly for specific information between different runtime states. Furthermore, this filter functionality could be extended to allow advanced searches techniques that are not yet implemented in our interface.

```
1   <template>
2       ...
3       <!-- Displays information depending on the selected info (switch button) -->
4       <div v-if="compareInfo == 'facts'" v-for="(fact, index) in filteredFacts" :key="
            index">
5         {{fact.textual}}
6       </div>
7
8       <div v-if="compareInfo == 'actions'" v-for="(action, index) in filteredActions"
            :key="index">
9         {{action.textual}}
10      </div>
11
12      <div v-if="compareInfo == 'duties'" v-for="(duty, index) in filteredDuties" :
            key="index">
13        {{duty.textual}}
14      </div>
15
16      <div v-if="compareInfo == 'violation'" v-for="(violation, index) in
            filteredViolations" :key="index">
17        <div v-if="violation['violation']=='invariant'">Violated invariant: {{
              violation['invariant']}} </div>
18        <div v-else-if="violation['violation']=='action'">Violated: {{
              violation.value.textual}} </div>
19        <div v-else >Violated: {{violation['value']['textual']}} </div>
20      </div>
21      ...
22  </template>
23
24  <script>
25  export default {
26      ...
27      props: {
28          ...
29          // text to search on (search input)
30          searchVar: String,
31          // type of info to compare with each other (switch button)
32          compareInfo: String,
33          // node identifier of the column
34          nodeID: Number
35          ...
36      },
```

```
37      data: function () {
38          return {
39            edges: [],
40          }
41      },
42      mounted() {
43          // retrieves the execution trace (Listing 4.6) for the column
44          this.edges = this.getStateEdges(this.nodeID)
45      },
46      // Functions to search/filter information depending on the search input field
47      computed: {
48          filteredFacts() {
49              return this.edges[0].target_contents.filter(fact => {
50                  return fact.textual.toLowerCase().includes(
                        this.searchVar.toLowerCase())
51              })
52          },
53          filteredActions() {
54              return this.edges[0]['all-enabled-transitions'].filter(action => {
55                  return action.textual.toLowerCase().includes(
                        this.searchVar.toLowerCase())
56              })
57          },
58          filteredDuties() {
59              return this.edges[0]['all-duties'].filter(duty => {
60                  return duty.textual.toLowerCase().includes(
                        this.searchVar.toLowerCase())
61              })
62          },
63          filteredViolations() {
64              return this.edges[0]['violations'].filter(violation => {
65                  if (violation['violation'] =='invariant') {
66                      return violation.invariant.toLowerCase().includes(
                            this.searchVar.toLowerCase())
67                  } else {
68                      return violation.value.textual.toLowerCase().includes(
                            this.searchVar.toLowerCase())
69                  }
70              })
71          },
72      },
73  }
74  </script>
```

**Listing 4.7: Implementation of the filter functionality for eFLINT in Vue and JavaScript. Each column in the inspect modal contains the following Vue and JavaScript code**

## 4.6 Generic design

From our implementation, we can see that a lot of the code that interacts with the exploring interpreter back-end is generalized. Most language-dependent code is code in how we display information for the specific language. It takes relatively low effort to implement our computational design for another language. In figure Figure 4.1, we show an example of a computational notebook that has been implemented for the Idris language (a purely-functional programming language) with the same exploratory programming functionality that our eFLINT implementation provides. The execution graph of the example is illustrated in Figure 4.2.

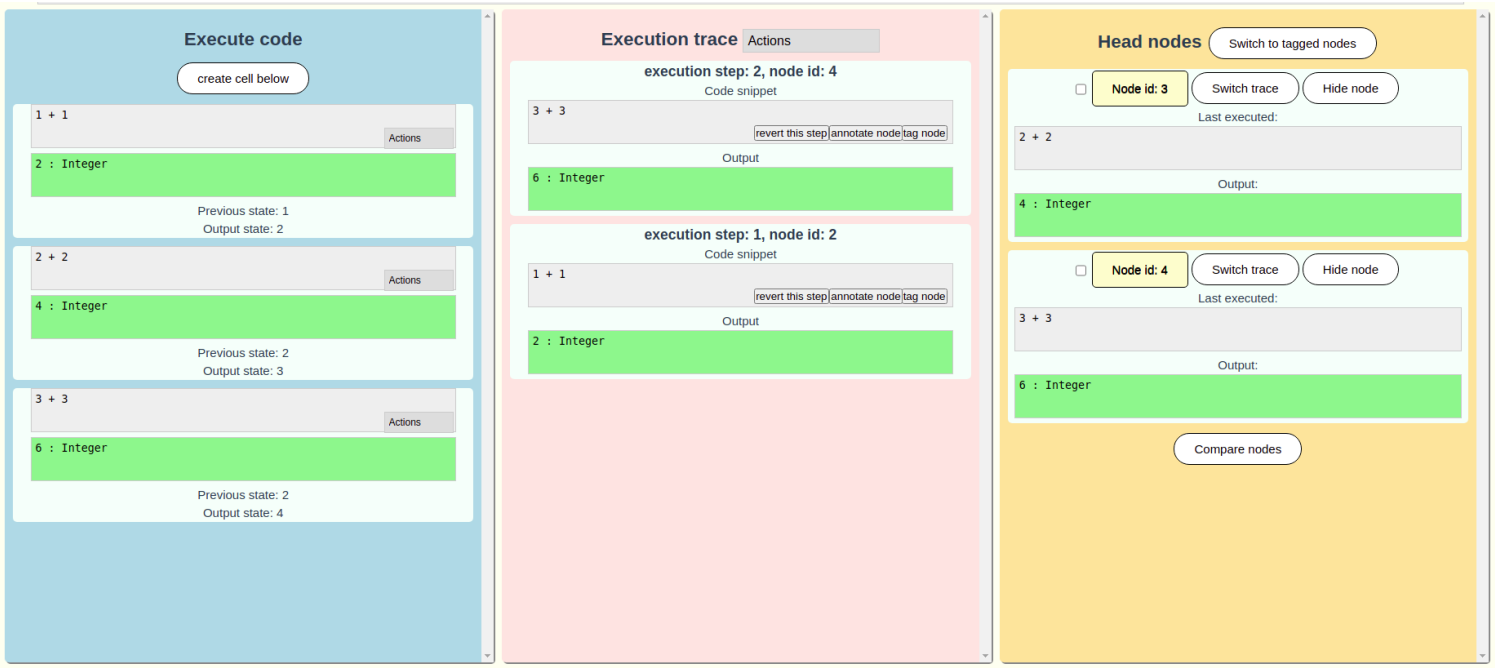



**Figure 4.1: A computational notebook on top of an exploring interpreter implemented in the Idris language.**

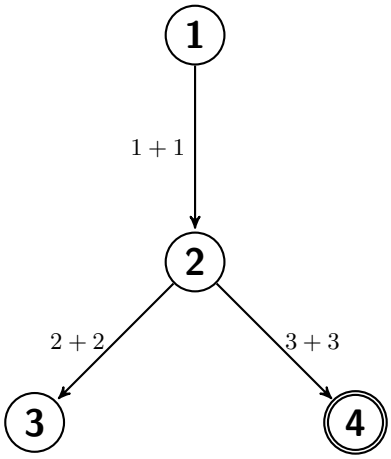



**Figure 4.2: The execution graph of the computational notebook in Figure 4.1.**

# Chapter 5

# Evaluation

For the evaluation of our computational notebook on top of an exploring interpreter, we will make use of a usability study. We do this to gain concrete knowledge and insights about our design and implementation. In the usability study, we will try to evaluate the practicality and show how an 'exploring interpreter' can help the lack of support for recording and sensemaking of exploration history and experimentation for exploratory programming in computational notebooks. The usability study will be done by comparing our computational notebook on an exploring interpreter with JupyterLab, the next-generation web-based user interface for Project Jupyter. Furthermore, we will go into the limitations of our usability study by analyzing the threats to the validity of our evaluation at the end of this chapter.

During our experiments, we will compare the support for recording and sensemaking of exploration history and the ease of experimentation between the two notebooks by asking questions in the context of exploratory programming to participants and whether they are able to complete specific exploratory programming tasks. In order to do so, participants will prototype and experiment with different ideas to develop distinct programs in each experiment. As such, the study is designed to answer our research questions stated in the introduction:

- RQ1) How practical are exploring interpreters for building computational notebooks to support exploratory programming tasks?
- RQ2) How can we better support exploratory programming in computational notebooks?
  - RQ2a) To what extent does an 'exploring interpreter' solve the lack of support for experimentation in computational notebooks in the domain of exploratory programming?
  - RQ2b) To what extent does an 'exploring interpreter' solve the lack of support for recording and sensemaking of exploration history for computational notebooks in the domain of exploratory programming?

We conduct the experiments with different sequential programming languages and code snippets in JupyterLab and our computational notebook. In Jupyter Notebook, the participant will make use of Python3. Alternatively, in our computational notebook, the subjects will program with eFLINT. The code cells in JupyterLab consist of simple math operations, while the notebook on top of an exploring interpreter contains additions and deletions of facts.

The experiments are carried out among a group of participants who have experience with Python3. The group consists of programmers with various experiences using the eFLINT language and computational notebooks in general. A total of six people took part in the experiment where four people had experience with JupyterLab, and three participants had programmed with eFLINT before.

Initially, we give the participants specific instructions to familiarize them with the language and notebook. In each subsequent experiment, the subjects get more freedom to prototype and experiment with different ideas. During the experiments, we try to not steer participants to our expected answers by omitting the problems of computational notebooks in the experiments. We do this done to let them think about the problems of computational notebooks in the context of exploratory programming themselves while carrying out the tasks and what an exploring interpreter can offer.

## 5.1 Experiments

In this section, we will briefly explain the experiments that we have carried and their results. Full details of the whole experiment can be found in Appendix A, with the answers of each participant in Appendix B.

### 5.1.1 Experiment 1

In the first experiment, a programmer will be tasked to create three different explorations by executing a number of code cells out of order. Afterwards, the programmer will consistently switch between the three different explorations and execute another existing code snippet. This process will be repeated until it reaches a certain number of iterations or when the programmer is unable to recover a past exploration.

At the end, the programmer will be asked to compare the different explorations and switch to the exploration that has the highest variable value (in JupyterLab) or the exploration that has the most amount of facts that are true (in the notebook on top of an exploring interpreter). Furthermore, the participant will be tasked to use the exploration history in order to recall effects of some of the code snippet that he has executed.

### 5.1.2 Experiment 2

In the second experiment, a programmer will start one exploration by executing a set of code cells. Next, the programmer will be asked to edit and make some changes in some of the code cells, which are then executed again. Afterwards, the programmer will be requested to freely try out ways to undo or revert several code snippets to experiment with different ideas.

Just like the previous experiment, the programmer will be asked to compare the different explorations and switch to the exploration that has the highest variable value (in JupyterLab) or the exploration that has the most amount of facts that are true (in the notebook on top of an exploring interpreter). Furthermore, the participant will be tasked to use the exploration history in order to recall the effects of some of the code snippet that he has executed.

### 5.1.3 Experiment 3

In the last experiment, we gave the participant the freedom to prototype and experiment with the two computational notebooks. However, we did put some constraints that they should use the computational notebooks as a tool for exploratory programming. We hope that the participants can recognize the exploratory programming problems in current computational notebooks and how a notebook on top of an exploring interpreter can alleviate these problems. Furthermore, we want to discover whether participants are able to discover other potential uses of the computational notebook on top of an exploring interpreter.

## 5.2 Results

Participants in the following section will be noted as P1 to P6.

### 5.2.1 Completing exploratory programming tasks

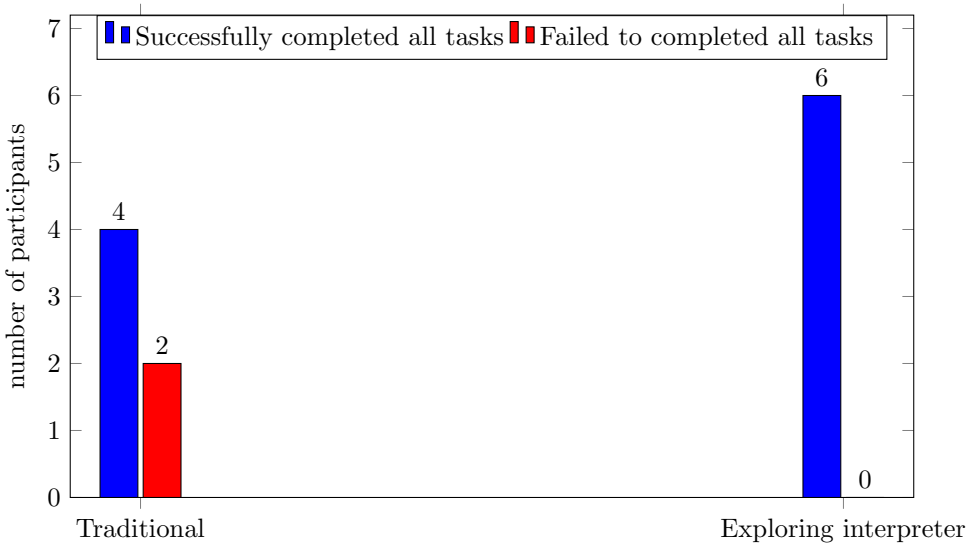



**Figure 5.1: Number of participants that successfully or failed to complete all exploratory programming tasks between a traditional computational notebook and a computational notebook on top of an exploring interpreter.**

With the computational notebook on top of an exploring interpreter, participants were able to easily complete all exploratory tasks as seen in Figure 5.1. However, not every participant was able to complete all given tasks in JupyterLab. Furthermore, all participants that succeeded in completing the tasks in JupyterLab mentioned that they came across a lot of trouble or had to resort to various hacks. P1 mentioned that:

*"To be able to restore three distinct checkpoints at will, it seemed like I had to make three distinct notebooks (or the save button would just overwrite my checkpoint). Once I had three side by side, I could switch tabs to compare their outputs" - P1*

Additionally, P3 stated that the traditional notebook forced him to keep track of his approaches either in his head or by writing the alternatives somewhere else (e.g., piece of paper).

Lastly, P4 had a lot of difficulties in maintaining a complete image of the explorations in his head when he worked with JupyterLab, and he felt that traditional notebooks did not have the support for these kinds of exploratory programming tasks.

### 5.2.2 Recovering (past) explorations

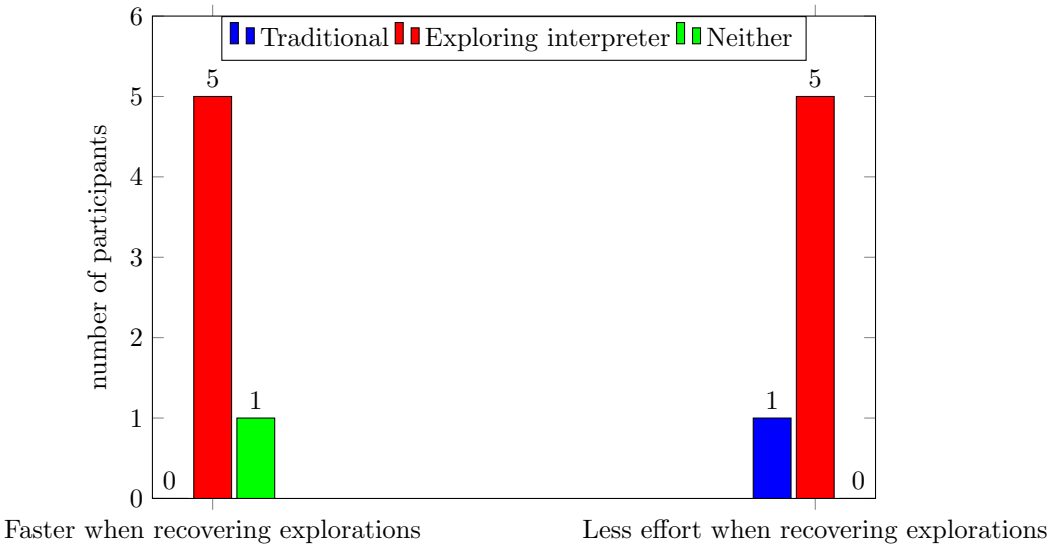



**Figure 5.2: Number of participants that answered whether a traditional notebook or a computational notebook on top of an exploring interpreter was faster and took less effort when recovering past explorations.**

As seen in Figure 5.2, most participants answered that the computational notebook on top of an exploring interpreter was faster and took less effort during the recovery of past explorations. They stated that our computational notebook supported an easy way to navigate between the different explorations using the switch trace button in the head nodes without having to keep track of the cells to be executed themselves, save explicit checkpoints or tediously repeat steps. In JupyterLab, participants mentioned that the notebook only allows one exploration session and that Jupyter checkpoints were not sufficient for recovering multiple explorations.

*"In JupyterLab, finding the checkpoints and reverting them is an error-prone task" - P4*

*"With an exploring interpreter, it is easier and faster to recover a past exploration. It has a more clear interface showing the past execution and provides buttons to revert to the desired step. In the Jupyter notebook it is more difficult to do the same thing because we can only mark the revert place after one complete exploration. Then we have to repeat the experiment to mark the revert point" - P5*

Moreover, P1 brought up the fact that our notebook always guaranteed a way to recover a correct past state of the notebook that they visited even if it could become confusing. In JupyterLab, P1 had to either create a lot of checkpoints and overwrite states carefully. which took a lot of effort:

*"Both notebooks could both get quite confusing because I still had to model the reasoner's state in my head, but at least in the case of eFLINT, I had confidence that every exploration really did take me back to SOME correct previous state (where in Jupyter I had to either create loads of checkpoints, or overwrite the state carefully)" - P1*

However, two participants (P2 and P3) mentioned that the traditional notebook could be better or equal for simple exploratory programming tasks. P1 declared that he took the same amount of time for recovering explorations in JupyterLab and our computational notebook when there were only two exploration sessions. P2 brought up the fact that it took less effort in the JupyterLab interface due to the required effort to understand our notebook interface. Nonetheless, both participants mentioned that this was only true for simple tasks. For more complex or challenging tasks, they stated that a computational notebook on top of an exploring interpreter could have an advantage over traditional notebooks because users would need a way to keep track of every action that is made when tasks become more difficult or complex.

*"When there are only two exploration sessions, timing is identical. When there are more, the exploring interpreter makes it much faster via a button click instead of having to execute the whole session again" - P2*

*"Considering the required effort to understand the notebook interface on top of an exploring interpreter, I think it takes less effort to use the traditional notebooks when recovering past explorations. However, this is true for simple tasks. Perhaps for other, more challenging tasks (e.g., data analysis), this can be different because users need to keep track of each of their actions" - P3*

### 5.2.3 Undoing changes

During experiment 2, all participants (P1-P6) faced challenges when trying to undo changes to programs in JupyterLab. They stated that it was not easy to undo executions and that they had to either re-execute cells or revert to checkpoints in order to reproduce previous results or notebook states in JupyterLab. On the other hand, in our computational notebook, every participant had the same opinion that undoing changes were really natural and better supported than in JupyterLab.

P1 and P5 relied on the checkpoint functionality of JupyterLab to undo changes, where they had to be very careful in overwriting the checkpoints. Both participants and P3 mentioned that the tracking of explorations in different nodes with the dedicated switch button contributed a lot to the difference in experience between the two notebooks:

*"The eFLINT notebook tracks every single action ever and makes it possible to revert to that moment. Reverts are also non-destructive, because it creates a NEW sequence of actions, while still keeping the OLD. Jupyter's checkpoint system is very limited because (a) I have to explicitly do it myself, (b) its very hard not to overwrite a previous checkpoint, and (c) when I work hard to have multiple checkpoints, the relationship between these checkpoints is not obvious. The biggest contributing factor to eFLINT being better at this is the fact that its version of checkpoints (nodes) don't overwrite each other, and the sequence between them is tracked explicitly." - P1*

*"In Jupyter I always need to pay attention to each execution of the code cell because it is not easy to undo the execution. Most of the time I need to reproduce the results by reverting to checkpoint and re-execute some more cells if I did not update the check point. The exploring interpreter makes it easier to do so because I can decide the revert point after I execute all the cells. Since it also provides the execution trace, I can easily select the step I would like to revert to. Besides, the "%history" command shows all executions, including the ones I skip by reverting, so it makes it difficult to recognize the operations I add in a new exploration" - P5*

*"I think that in a notebook on top of an exploring interpreter, it is easier to execute undo actions than in a traditional notebook because there is a dedicated button for this purpose. While in the traditional notebook, users have to restart the kernel or re-execute cells, which might lead the overall notebook to become inconsistent if users try to run all the cells from top-to-bottom. Although traditional notebook offers workarounds to achieve the same results, I think the approach provided by notebook on top of an exploring interpreter is more straightforward to use despite the UI design" - P3*

Finally, P4 relied a lot on the different revert functionality provided by the computational notebook on top of an exploring interpreter to undo changes:

*"The undo/revert functionality was better supported in the environment with the exploring interpreter. It was cumbersome to explore this functionality in JupyterLab. It feels like the environment with the exploring interpreter was designed to support this functionality: having the three different views and the possibility to navigate different explorations made the whole experience smoother" - P4*

### 5.2.4 Recording and sensemaking of exploration history

In JupyterLab, many users had difficulties in using the exploration history to compare explorations or to recall effects of code snippets compared to the notebook on top of an exploring interpreter. P1, P2, P3, P4 and P5 mentioned that comparing explorations in our computational notebook was trivial to do.

However, in JupyterLab, they had to re-execute the session multiple times, which requires remembering the whole session:

*"Comparing traces and effects was much better with eFLINT since running the same action again creates a NEW node+output, while running the same Jupyter cell again REPLACES any existing output (if you want to compare both, you need to add new files side-by-side, or recreate previous steps in new cells)" - P1*

*"Jupyter only allows one exploration session and reverting loses the other session. This makes it really difficult to compare different sessions. The exploring interpreter notebook allows many sessions and makes comparing trivial. Furthermore, Jupyter required re-executing the session to see the results, which is error prone" - P2*

*"The undo option in notebook on top of an exploring interpreter allows me to compare different explorations easily. Instead, traditional notebook force me to keep track of the alternatives either in my head or by writing them somewhere else (e.g., piece of paper)" - P3*

*"The exploration in a normal notebook is difficult and prone to be lost. Whereas the on notebook on top of an exploring interpreter supports an easy navigation among the different explorations to compare them" - P4*

*"In the notebook with exploring interpreter, the explorations are recorded in different nodes and they can be retrieved by simply clicking the "switch trace". It is really convenient when checking explorations" - P5*

As for checking exploration and recalling the effect of operations, most participants also agreed that the exploring interpreter was able to support better this than traditional notebooks. P1, P2, P4 and P5 stated that the stack-like execution trace of our notebook gave enough information for the user to read the effects of each step easily, while the traditional notebook required users to re-execute sessions again to check their effects:

*"In JupyterLab, the printed history allowed me to recreate any trace I liked in a new cell from start to finish and see all outputs in sequence. It's not very handy but it works. However, in the exploring interpreter notebook, the fact that each trace is stack-like, showing results at each step means I can just read over it" - P1*

*"It was easier to understand the effects of each operation when working with the environment with the exploring interpreter. You could easily pinpoint what action impacted the data" - P4*

*"Checking the effect of each operation is easier when using a notebook with the exploring interpreter because the output of each step is displayed in the trace. Instead, in the Jupyter, I always need to execute print(x) to check the value of x after I make changes to it. I like the tag function and the function to inspect trace. They are useful for debugging and to verify the traces. I also like that in the middle column it shows the result of the execution of the cell so I can easily see the effect of them" - P5*

However, P3 answered differently compared to the other participants. P3 thinks that both computational notebooks have different strengths and weaknesses when recalling effects of operations:

*"On the one hand, the notebook on top of an exploring interpreter allows me to check every exploration quickly. However, recalling the effect of each operation was difficult because there are no explicit hints about how the elements in each of the three tabs relate to each other. On the other hand, in traditional notebook, it is harder to achieve this task, because essentially one has to run and restart the kernel several times to get different explorations, but the results and how each result is related to each other is precise (assuming a re-arrangement of cells to guarantee a linear execution)" - P3*

### 5.2.5 Overall thoughts

For exploratory programming, most participants stated that the computational notebook on top of an exploring interpreter provides many features that they missed in JupyterLab.

P1 found it very intuitive to reason about the exploration of the program as a tree of actions/steps, and he thinks that the nodes and trace approach is the right direction for exploratory programming in computational notebooks. P1 expressed that:

*"Exploratory programming with Jupyter was very difficult because the only way to understand the FULL trace in a particular state was to use the history, which kinda defeats the purpose of using a notebook. I missed the node list" - P1*

P2 noted that JupyterLab felt really restricted and did not support exploratory programming as much compared to our computational notebook due to the missing revert and compare functionality.

Furthermore, P4 argued that the programming environment with the exploring interpreter fits the need to perform exploratory programming because Jupyter is more focused on keeping a linear narrative. On the other hand, the exploring interpreter allows the user to create different sequences and readings of the same code. Additionally, the participant mentioned that:

*"JupyterLab misses the two additional views created in the environment with the exploring interpreter. Being able to switch among explorations and visualizing the order and output of each operation helps tremendously in the exploratory process. In general, it saves time when trying to understand the output of a given execution" - P4*

However, P4 felt that the main narrative of the computational notebook migrates to the execution trace view because the notebook with code cells gets less attention, where the notebook is used as an environment to write any code and all the narrative is kept in the exploration trace. P4 mentioned that it would be nice to connect these two views in such a way that the notebook still has an important role within the exploration.

P5 declared that the notebook on top of an exploring interpreter provides many features for trace recording and comparing that Jupyter does not have for exploratory programming. According to the participant, the computational notebook on top of an exploring interpreter is more suitable for non-experienced programmers or more complex tasks while the JupyterLab is more appropriate for experienced programmers:

*"The notebook on top of an exploring interpreter provides many features for trace recording and comparing that Jupyter does not have. I can imagine how a notebook on top of an exploring interpreter can help for exploratory programming when the codes have a proper complexity or when one starts learning programming. Jupyter is more suitable for people who have some experiences of programming and thus have a clear picture of their code traces" - P5*

Finally, P3 claimed that the exploring interpreter offers interesting features that might be valuable for exploratory programming tasks, but its usage depends on various factors:

*"Exploratory programming is a complex field, mainly because it depends on each domain and user's background. Moreover, some experience using traditional notebook might affect how users perceive an notebook on top of an exploring interpreter. Therefore, for creating and comparing different explorations, I think notebook on top of an exploring interpreter does a better job than notebook on top of an exploring interpreter" - P3*

#### Integrating exploring interpreters into existing computational notebooks

Although many participants preferred the computational notebook on top of an exploring interpreter as support for exploratory programming, some participants recommended that the implemented functionality could be added to existing computational notebooks. They argued that the exploring interpreter could be integrated within Jupyter notebooks in order to solve the underlying problems of traditional computational notebooks while maintaining the existing advantages.

*"I think the underlying technology of the notebook on top of an exploring interpreter can be integrated*

*within Jupyter notebooks. This might help users identify the benefits of having an exploring interpreter behind a notebook and improve the look and feel of the notebook on top of an exploring interpreter" - P3*

*"I think that the support of the exploratory programming is well supported on the environment with the exploring interpreter. However, I feel that this can be added to JupyterLab in order to have a complete and mature environment that draws on the capabilities and nice user experience of JupyterLab" - P4*

### 5.2.6 Unused or rarely used functionality

Many users mentioned that some functionalities were rarely used or not at all, which was most of the time argued to be caused by the simple exploratory programmings tasks:

*"No inspection wasn't used, because the trace view gave all information" - P2*

*"I think there are several functionalities that I did not try (e.g., switch to tagged nodes, inspect nodes), mainly because of the lack of knowledge with the tool, and the experiments were simple, so it was not necessary to use them" - P3*

*"I did not explore the other options displayed in the head nodes view (e.g. inspect node)" - P4*

*"I did not use the switch trace a lot and I think the reason may be that the example case was not too complicate. The query result and effects of execution can be seen easily in the middle column so I don't really need to make many different traces to compare the difference. Change cell version was not used a lot either because I don't have to make too much changes for the code; I can just reuse the cell to produce different outputs" - P5*

## 5.3 Discussion

### 5.3.1 Viscosity

From our results in subsection 5.2.2 and Figure 5.2, we have seen how most participants had an easier time (faster and less effort) in recovering past explorations using the computational notebook on top of an exploring interpreter compared to the traditional notebook. Furthermore, in subsection 5.2.3, we have observed that all participants had less trouble in undoing changes to the program in our computational notebook.

These differences are mostly attributed to the dedicated switch button that provided a clear way for users to revert to another notebook state or undo changes that are made to a program that traditional notebooks do not have. A computational notebook on top of an exploring interpreter tracks every single action of a user and makes it possible to instantly revert to a specific node (notebook state) at any moment.

To answer RQ2a in section 1.1, an exploring interpreter could lower the viscosity in computational notebooks due to its clear revert functionalities. By lowering the viscosity, we can improve the ease of experimentation in a computational notebook and reduce the risk of making exploratory risks:

> **Finding 1:** A computational notebook on top of an exploring interpreter may have a lower viscosity than traditional computational notebooks due to its clear revert functionality and thus improve the support for experimentation.

### 5.3.2 Provenance work

In subsection 5.2.4, we discovered that participants had an easier time understanding the effect of each operation and comparing different sessions in a computational notebook on top of an exploring interpreter. Furthermore, it was easier for users to see how a notebook had entered a given state.

These results are most likely a consequence of the execution graph that is made available by the exploring interpreter. The execution graph makes it possible to record every new code snippet that has been tried out in a new node (configuration) in the execution graph with its effect on the notebook state.

This recording of explorations in nodes made it possible for a user to easily compare different nodes by switching traces or using the inspect modal. In JupyterLab, rerunning cells in Jupyter would replace existing output which made it more difficult to compare different sessions with other.

Additionally, the execution trace that we have presented in the middle column of our notebook contributed a lot to the support for recording and sensemaking of the exploration history. The stack-like trace is able to capture the execution order of computational notebooks with the effects of each execution step automatically. In traditional notebooks, a user would have to collect effects on the notebook by printing the value with a different function manually.

Regarding RQ2b in section 1.1, an exploring interpreter could provide an automatic recording of provenance work (configurations) in a computational notebook. This result could improve the support for recording and sensemaking of exploration history such that users are able to reproduce explorations better, compare past explorations or recall past effects of code snippets:

> **Finding 2:** A computational notebook on top of an exploring interpreter may provide a more effective way of collecting provenance work than traditional computational notebooks due to the automatic recording of configurations and thus improve the support for recording and sensemaking of exploration history.

### 5.3.3 Practicality

Results of Figure 5.1 and subsection 5.2.5 indicate that a computational notebook on top of an exploring interpreter provides users with an easier experience when carrying exploratory programming tasks out when programmers need to create and manage multiple explorations or use the exploration history to reason about past approaches.

This outcome is most likely contributed by finding 1 and finding 2. First of all, the improved support for experimentation helped programmers to easily work on several explorations simultaneously without redoing everything with the help of the revert functionalities. On top of that, the improved support for recording and sensemaking of exploration history with the execution graph and execution trace proved to reduce a lot of tedious manual work involving collecting provenance work for comparing explorations and recalling effects of previous code snippets.

As for RQ1 in section 1.1, an exploring interpreter can provide many functionalities that can make computational notebooks more practical for exploratory programming:

> **Finding 3:** A computational notebook on top of an exploring interpreter could be more practical than traditional computational notebooks for exploratory programming tasks that involve creating and managing different explorations, comparing past explorations or recalling effects of operations due to the additional features that an exploring interpreter provides.

### 5.3.4 Flaw

Users of our computational notebooks mentioned a flaw for the computational notebook on top of an exploring interpreter. As stated in subsection 5.2.2 subsection 5.2.2, P1 and P3 sometimes found our notebook hard to use, which was mostly due to our user interface design decisions of the computational notebook. Furthermore, during the experiment, it was not thoroughly explained how a user could interact with the elements in each of the columns, and it could be difficult to see how each column was connected to each other.

However, despite the flaw of our computational notebook, all participants are in agreement that our computational notebook is able to support exploratory programming tasks better than JupyterLab in our experiments, as we have concluded in finding 3.

> **Finding 4:** Despite the required effort to understand and use the computational notebook on top of an exploring interpreter, the experiments suggest that our computational notebook is more practical than JupyterLab.

### 5.3.5 Debugging potential

Our computational notebook was also of use for debugging purposes. As stated by P5 in subsection 5.2.4: "I like the tag function and the function to inspect trace. They are useful for debugging and verifying the traces". Tagging allowed users to save specific notebook states and analyze specific breakpoints of traces. Furthermore, the execution trace can serve as a way to see what action impacted which data to understand when something went wrong easily during a task.

**Finding 5:** The execution trace and the revert functionality provided by the exploring interpreter may present users with alternative mechanisms to debug computational notebooks during exploratory programming.

### 5.3.6 Confidence

In subsection 5.2.2, P1 stated that he always had confidence that our computational notebook was able to recover some correct previous state. This result is an extremely relevant remark about confidence. Users will rely more and more on the notebooks when some confidence is established rather than on their mental model over time (assuming confidence is maintained).

The ability to revert to any state in the execution graph without the need to rerun any code is a major factor that caused this result. In JupyterLab, reverting to a previous exploration is error-prone because it requires a user to remember the exact execution steps as mentioned by P2 and P4 in subsection 5.2.2 and subsection 5.2.4.

**Finding 6:** Participants had more confidence in the computational notebook on top of an exploring interpreter than JupyterLab when carrying out exploratory programming tasks that involved switching between explorations.

## 5.4 Threats to validity

Our usability study contains several threats to validity. Firstly, our study focused primarily on the language *eFLINT* (computational notebook on exploring interpreter) and Python3 (JupyterLab). Furthermore, all participants of our usability study had experience with programming. Consequently, these factors can result in findings that may not be generalized to other languages, different computational notebooks or other types of programmers, such as novice programmers.

Secondly, due to the constraint of time, we could not conduct the study with a large number of participants over a long period. This situation means that we can not reliably claim that we have grounded results that are applicable to a large number of participants with our small sample size. The constraint of time also resulted in relative short programming sessions during each experiment. Some exploratory programming sessions in real situations are easily far beyond that. It is possible that our computational notebook could not fare in such scenarios due to the increasing amount of approaches and ideas. However, in subsection 5.2.2 and subsection 5.2.5, some participants did expect that our computational notebook will be more suitable for exploratory programming that involves long or complex exploratory programming tasks than JupyterLab.

Thirdly, exploratory programming is a complex field. There exist various factors that impact the support for experimentation and the support for recording and sensemaking of exploration history. In this thesis, we only looked at the collection of provenance work and the recovery of explorations. Ideally, we would like to measure other factors too in the experiments.

Fourthly, the experiments that we are using to evaluate our computational notebook does not encapsulate every usage of exploratory programming in notebooks but only a part. It could be possible that our evaluation does not hold in other experiments compared to a normal notebook. However, the goal of this thesis was to show the potential that notebooks on an exploring interpreter have in the domain of exploratory programming to solve existing problems.

Lastly, we designed questions to evaluate our computational notebook with JupyterLab. However, formulating completely neutral questions is difficult. It could be possible that the questions were too biased, which caused it to nudge participants to a specific answer.

# Chapter 6

# Limitations & Future work

In this chapter, we discuss the limitations of the computational notebook on top of an exploring interpreter in comparison with a current existing computational notebook, JupyterLab. Furthermore, we will analyze future work for further research.

## 6.1 Limitations of our computational notebook on top of an exploring interpreter

In section 3.2, we argued that a computational notebook on top of an exploring interpreter should not have a destructive property and sharing property. However, a computational notebook on top of an exploring interpreter without the destructive property and sharing property requires more space. Without the destructive property, the exploring interpreter will not remove nodes or edges whenever a user performs a 'revert' action. Without the sharing property, the computational notebook will always create a new configuration or node even if the resulting configuration or node after an 'execute' action already exist in the execution graph. On the other hand, most traditional notebooks do not keep track of every notebook version. Therefore, it is likely that our computational notebook is less efficient in space than traditional computational notebooks.

Currently, we keep track of every configuration or node as a list in the right column (head nodes) in the notebook. For simple exploratory programming tasks, this has been very useful for users to switch between explorations or compare different notebook versions with each other. However, the list of nodes can become unmanageable for complex or longer exploratory tasks. The user experience of our computational notebook is an aspect that needs more attention.

## 6.2 Future work

The promising results from the usability study indicate that further investigation into exploring interpreters for computational notebooks can be beneficial. Future work can focus on improving the current design or implementing additional functionalities that impact other factors of exploratory programming. For example, as stated by P1 in 5.2.5, the exploring interpreter enables participants to reason about the exploration of the program as a tree of actions/steps. In a future notebook version, we could make this treelike structure more explicit in the computational notebook and allowing users to inspect the tree visually or interact with it. Furthermore, we can design more interactions that are possible with the execution trace, such as a way to re-execute the whole execution trace with modified nodes in the path.

In our usability study, we have only compared our computational notebook with JupyterLab, as discussed in section 5.4. For a future study, it could be important to do a usability study with different computational notebooks in order to investigate whether our results and findings are still valid. Furthermore, the usability study could be expanded by implementing the computational notebook on top of an exploring interpreter with other programming languages. Additionally, we would like to expand the usability study by inviting a larger number of participants, increasing the duration of experiments and creating more complex exploratory programming tasks.

In this thesis, we focused on two different aspects of exploratory programming in computational notebooks. However, as mentioned in section 2.1 and subsection 5.2.5, exploratory programming is a complex field that involves more than just the two characteristics, mainly because it depends on

each domain and user's background. In a future study, we would like to analyze and compare other characteristics of exploratory programming for computational notebooks that we have not discussed in this thesis.

During our experiments, we noticed that participants did not know that there were features possible while doing the exploratory programming tasks. This problem could have been prevented if we explicitly instructed users to use them, improved the user interface design or explained the functionality more in detail.

Furthermore, as observed in subsection 5.2.5, some features of our notebook have been used rarely or not at all. For example, most participants did not utilize the inspect functionality or search functionality due to the lack of knowledge about the tool or because the functionality was not necessary for the simple tasks that we have created. In future experiments, we could solve this issue by designing a more difficult experiment and by giving demonstrations of the functionality to evaluate whether these functionalities are useful.

Lastly, as argued by the participants of our usability study in subsection 5.2.5, an exploring interpreter could be integrated into traditional computational notebooks to solve current exploratory programming problems. Therefore, a study is needed to explore and evaluate the integration of exploring interpreters into traditional computational for exploratory programming.

# Chapter 7

# Related work

In current literature, there are several tools that have been developed in order to tackle the lack of tool support recording and sensemaking of exploration history and experimentation during exploratory programming in computation notebooks. In this chapter, we will briefly go over some of these tools and compare their work with ours.

## 7.1 Versioning

Previous studies [8, 29] have argued that exploratory programming benefits from version control to reproduce previous explorations such that it is easier to experiment with different approaches.

However, as mentioned in subsection 2.4.1, current version control tools for computational notebooks are not sufficient. Chattopadhyay *et al.* explained that traditional versioning systems have been found to not been effective when versioning computational notebooks to support recording of explorations [8]. In their study, data scientists stated that "using traditional versioning mechanisms intended for source code are just a complete and utter failure" and that "the history and execution order of the notebook cannot be tracked by version control systems". This means that programmers are not always able to successfully reproduce a previous state of the notebook.

Kery *et al.* introduce variolite, which is a lightweight tool for local versioning of the code of a file [10]. The tool provides users with a way to create boxes around different sections in a file which enables them to create alternatives of functions or a single line of code. According to the authors, this tool supports nonlinear exploration by enabling the exploration of different combinations of alternatives that they have tried. Verdant is a system built after Variolite by the same authors of Variolite that records versions of artifacts in notebooks and their reproduction steps. They argue that "Verdant provides light-weight interactions for comparing, replaying, and tracing relationships among many versions of different code and non-code artifacts in the editor" [11].

In contrast, we support versioning of different runtime states of the notebooks where programmers can easily switch between the versions of notebooks to continue an exploration that a programmer went through in the past. In Variolite and Verdant, a programmer needs to re-run their code in correct order after to continue their exploration, which does not always successfully reproduce its previous result. However, our solution makes it possible to avoid re-running code by switching to a previous notebook state without much effort because each node in the execution graph contains the necessary information that the exploring interpreter needs to continue a past exploration.

## 7.2 Provenance

In current computational notebooks, most programmers need to track provenance manually. They either need to remember every step that they did in their head or write down their results on paper because it is easy to lose track of the steps that you have taken when cells are rerun and overwritten [19]. There are different tools or libraries that have been developed to collect provenance work in computational notebooks. However, most solutions are designed for programming languages such as Python [30–32] or specialized for data works [33, 34]. In our work, an exploring interpreter could be integrated within computational notebooks for any sequential language to enable provenance work that is tracked automatically.

## 7.3 Comparison within notebooks

During exploratory programming in computational notebooks, there exist different tools to track code history to enable comparison between versions of code fragments or output in notebooks. Head *et al.* created a set of code gathering tools that enables comparison between slices of code. Janus uses automatic versioning per code cells after execution [29] for comparison. Variolite [10] makes uses of versioning per code snippets, on the other hand, Verdant [11] does this for code and non-code artifacts.

Our approach with the exploring interpreter makes it possible to inspect different runtime states of the notebook with each other. This solution does not only compare different versions of execution traces with each other, but it is also possible to compare configuration information such as internal variable values that are not shown in the notebook. Furthermore, we allow programmers to compare execution logs with annotations of different explorations. Additionally, our approach includes a way for users to filter on specific information when comparing notebook states.

# Chapter 8

# Conclusion

Exploratory programming is the process of writing code by trial and error and is considered to be an important process of the software engineering cycle to try some designs to see what works in a new domain or when a requirement is not fully specified before implementation. Computational notebooks have seen an increase in use for exploratory programming due to their flexibility and accessibility. However, current computational notebooks suffer from a lack of support for experimentation and recording and sensemaking of the exploration history in order to fully support exploratory programming. In this thesis, we have developed a computational notebook on top of an exploring interpreter to tackle the problems of current computational notebooks. An exploring interpreter can provide useful functionalities for computational notebooks in the realm of exploratory programming that are not present yet.

The potential of using an exploring interpreter for the back-end of computational notebooks has been shown in several experiments with a usability study among a group of participants. Despite the required effort to understand the notebook interface on top of an exploring interpreter, most participants argued that our computational notebook is able to support exploratory programming better than current computational notebooks in the domain of experimentation and recording and sensemaking of the exploration history.

Firstly, the exploring interpreter can lower the viscosity of computational notebooks due to revert features that make it possible for users to switch easily among explorations or undo actions without much risk.

Secondly, the exploring interpreter provides an approach to collect provenance work in the execution graph automatically. The execution graph can be used to visualize the order and effect of each code snippet execution in nodes that can help programmers to reason and to compare the different states of the computational notebook.

Lastly, we have shown that the computational notebook on top of an exploring interpreter can be more practical than traditional computational notebooks for exploratory programming tasks when users need to create and manage multiple explorations or use exploration history to reason about past approaches.

Throughout the experiments, we have also seen that the exploring interpreter can also provide a different mechanism for users to debug their computational notebook. The execution trace and the switch trace functionality has been discovered to be helpful to see what went wrong during the course of experimenting.

In conclusion, an exploring interpreter may be a solution to various problems of computational notebooks when used for exploratory programming by developing notebooks on top of an exploring interpreter or integrating the exploring interpreter within current notebooks.

# Acknowledgements

During the writing of this thesis, I have received a lot of help and support. Firstly, I would like to thank my supervisor, Thomas van Binsbergen, for this thesis opportunity and his knowledge in the field of this thesis. Secondly, I would like to acknowledge the participants that took part in the experiments for their input and feedback. Lastly, I would like to thank my family and friends for their comments and encouragement.

# Bibliography


[1] M. B. Kery and B. A. Myers, "Exploring exploratory programming," in *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, IEEE, 2017, pp. 25–29.

[2] J. W. Tukey, *Exploratory data analysis*. Reading, MA, 1977, vol. 2.

[3] C. Hill, R. Bellamy, T. Erickson, and M. Burnett, "Trials and tribulations of developers of intelligent systems: A field study," in *2016 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, IEEE, 2016, pp. 162–170.

[4] A. A. Jabal, M. Davari, E. Bertino, C. Makaya, S. Calo, D. Verma, A. Russo, and C. Williams, "Methods and tools for policy analysis," *ACM Computing Surveys (CSUR)*, vol. 51, no. 6, pp. 1–35, 2019.

[5] Y. S. Yoon and B. A. Myers, "A longitudinal study of programmers' backtracking," in *2014 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, IEEE, 2014, pp. 101–108.

[6] T. Kluyver, B. Ragan-Kelley, F. Pérez, B. Granger, M. Bussonnier, J. Frederic, K. Kelley, J. Hamrick, J. Grout, S. Corlay, P. Ivanov, D. Avila, S. Abdalla, and C. Willing, "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, 2016, pp. 87–90.

[7] R. D. Peng, "Reproducible research in computational science," *Science*, vol. 334, no. 6060, pp. 1226–1227, 2011.

[8] S. Chattopadhyay, I. Prasad, A. Z. Henley, A. Sarma, and T. Barik, "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*, 2020, pp. 1–12.

[9] A. Head, F. Hohman, T. Barik, S. M. Drucker, and R. DeLine, "Managing messes in computational notebooks," in *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, 2019, pp. 1–12.

[10] M. B. Kery, A. Horvath, and B. A. Myers, "Variolite: Supporting exploratory programming by data scientists.," in *CHI*, vol. 10, 2017, pp. 3 025 453–3 025 626.

[11] M. B. Kery and B. A. Myers, "Interactions for untangling messy history in a computational notebook," in *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, IEEE, 2018, pp. 147–155.

[12] P. Rein, J. Lincke, S. Ramson, T. Mattis, and R. Hirschfeld, "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*, 2017, pp. 17–27.

[13] M. Brachmann and W. Spoth, "Your notebook is not crumby enough, replace it," in *Conference on Innovative Data Systems Research (CIDR)*, 2020.

[14] D. Koop and J. Patel, "Dataflow notebooks: Encoding and tracking dependencies of cells," in *9th {USENIX} Workshop on the Theory and Practice of Provenance (TaPP 2017)*, 2017.

[15] L. T. Van Binsbergen, M. Verano Merino, P. Jeanjean, T. Van der Storm, B. Combemale, and O. Barais, "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*, 2020, pp. 84–100.

[16] B. Sheil, "Datamation®: Power tools for programmers," in *Readings in artificial intelligence and software engineering*, Elsevier, 1986, pp. 573–580.

[17] T. D. LaToza and B. A. Myers, "Hard-to-answer questions about code," in *Evaluation and Usability of Programming Languages and Tools*, 2010, pp. 1–6.

[18] T. R. G. Green and M. Petre, "Usability analysis of visual programming environments: A 'cognitive dimensions' framework," *Journal of Visual Languages & Computing*, vol. 7, no. 2, pp. 131–174, 1996.

[19] A. Rule, A. Tabard, and J. D. Hollan, "Exploration and explanation in computational notebooks," in *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, 2018, pp. 1–12.

[20] MATLAB, *version 9.0.0 (R2016a)*. Natick, Massachusetts: The MathWorks Inc., 2016.

[21] RStudio Team, *Rstudio: Integrated development environment for r*, RStudio, Inc., Boston, MA, 2015. [Online]. Available: `http://www.rstudio.com/`.

[22] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, *et al.*, "Apache spark: A unified engine for big data processing," *Communications of the ACM*, vol. 59, no. 11, pp. 56–65, 2016.

[23] *How to teach computational thinking*, `https://writings.stephenwolfram.com/2016/09/how-to-teach-computational-thinking/`, Accessed: 2021-05-05.

[24] M. B. Kery, M. Radensky, M. Arya, B. E. John, and B. A. Myers, "The story in the notebook: Exploratory data science using a literate programming tool," in *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, 2018, pp. 1–11.

[25] J. Notebook, "Ux survey results. h ps," *github.com/jupyter/surveys/blob/master/surveys/2015-12-notebook-ux/analysis/report dashboard.ipynb*, 2015.

[26] J. Freire, D. Koop, E. Santos, and C. T. Silva, "Provenance for computational tasks: A survey," *Computing in Science & Engineering*, vol. 10, no. 3, pp. 11–21, 2008.

[27] L. T. van Binsbergen, L.-C. Liu, R. van Doesburg, and T. van Engers, "Eflint: A domain-specific language for executable norm specifications," in *Proceedings of the 19th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, 2020, pp. 124–136.

[28] L. Bergroth, H. Hakonen, and T. Raita, "A survey of longest common subsequence algorithms," in *Proceedings Seventh International Symposium on String Processing and Information Retrieval. SPIRE 2000*, IEEE, 2000, pp. 39–48.

[29] A. C. Rule, *Design and use of computational notebooks*. University of California, San Diego, 2018.

[30] C. Bochner, R. Gude, and A. Schreiber, "A python library for provenance recording and querying," in *International Provenance and Annotation Workshop*, Springer, 2008, pp. 229–240.

[31] J. F. N. Pimentel, V. Braganholo, L. Murta, and J. Freire, "Collecting and analyzing provenance on interactive notebooks: When ipython meets noworkflow," in *7th {USENIX} Workshop on the Theory and Practice of Provenance (TaPP 15)*, 2015.

[32] J.-L. R. Stevens, M. Elver, and J. A. Bednar, "An automated and reproducible workflow for running and analyzing neural simulations using lancet and ipython notebook," *Frontiers in neuroinformatics*, vol. 7, p. 44, 2013.

[33] L. A. Carvalho, R. Wang, Y. Gil, and D. Garijo, "Niw: Converting notebooks into workflows to capture dataflow and provenance.," in *K-CAP Workshops*, 2017, pp. 12–16.

[34] Y. Wu, J. M. Hellerstein, and A. Satyanarayan, "B2: Bridging code and interactive visualization in computational notebooks," in *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*, 2020, pp. 152–165.

# Appendix A

# Experiment environment

During the experiments, we will ask you several questions. Although the answers are anonymous, we may be going to use the answers as quotes in a thesis. If you continue this experiment, then we assume that you have permitted this.

## A.1 Introduction

Exploratory programming is the process of writing code by trial and error. It is considered to be an important process of the software engineering cycle to try some designs to see what works in a new domain or when a requirement is not fully specified before implementation. It is a practice that makes it easy and fast to interactively prototype or debug small code applications compared to standard edit-compile-run-debug practice. Examples of exploratory programming activities are:

- Rapidly prototype different design ideas that you can compare with each other and choose the best result.
- Quickly make changes in existing code to see its effects without spending a lot of time and trouble.

In the experiments, we will ask you to follow instructions that require you to do some programming tasks on two different computation notebooks. For each task indicated by 'T', please note whether you are able to successfully do it. A computational notebook is a programming environment that enables exploratory programming, which gives programmers the ability to freely mix code, documentation and output in one document.

The first computational notebook is called JupyterLab (web-based interactive development environment for Jupyter notebooks, code, and data) in which you will program in the Python3 language. The second computational notebook is developed on top of an exploring interpreter in the eFLINT language, a domain-specific language for formalizing norms. An exploring interpreter, shortly explained, is a bookkeeping algorithm that enables a programming environment to keep track of runtime states in an execution graph.

[Link to JupyterLab]

[Link to the notebook on top of an exploring interpreter]

[Link to eFLINT paper]

[Link to eFLINT preprint paper]

In section A.3, you can see an explanation of the different functionalities that have been implemented in the notebook on top of an exploring interpreter.

## A.2 Experiments

For each experiment, create a new notebook for both JupyterLab (select 'Try JupyterLab' and after it is ready loading, file → new → notebook) and the eFLINT notebook on top of an exploring interpreter (create a new notebook on the homepage by pressing 'add instance').

Before starting the experiments, please answer the following questions with yes or no:

- Q0a) I have experience with computational notebooks:
- Q0b) I have experience with eFLINT:

### A.2.1 Experiment 1

**JupyterLab**

Create 5 new code cells (press b while not in a code cell) with the following code snippets:

1. $x = 111$
2. $x- = 11$
3. $x* = 13$
4. $x/ = 17$
5. print(x)

In the context of our experiments, explorations are alternatives or approaches by combining different code snippets in various order. Create three different explorations with the following steps (In order to switch to another exploration, you need to either restart the kernel and execute all the cells in the exact order or make use of checkpoints in notebooks. To save a checkpoint, you click on the disk icon next to the plus symbol in the notebook tab. To restore a checkpoint, click on the revert button found in File → Revert notebook to Checkpoint):

1. Execute cell 1 (you can execute a cell by pressing 'ctrl+enter' when selecting a cell)
2. Execute cell 2 to 4 in any order once
3. Execute cell 5
4. Restart the kernel (kernel → restart kernel)
5. Execute cell 1
6. Execute cell 2 to 4 in any order once (a different order than step 2)
7. Execute cell 5
8. Restart the kernel (kernel → restart kernel)
9. Execute cell 1
10. Execute cell 2 to 4 in any order once (a different order than step 2 and 6)
11. Execute cell 5
12. Restart the kernel (kernel → restart kernel)

T1a) Now we have three different explorations. Repeat the following tasks until you can not recover an earlier exploration or when the number of repetitions exceeds three:

- Switch to the first exploration, execute an additional code cell from cells 2-4 to expand the exploration.
- Switch to the second exploration, execute an additional code cell from cells 2-4 to expand the exploration.
- Switch to the third exploration, execute an additional code cell from cells 2-4 to expand the exploration.

Do the following tasks if possible:

- T1b) Compare the three different explorations and switch to the exploration with the highest value of x and print it by executing cell 5
- T1c) Check every exploration that you have attempted and try to recall effects of some operation that you have executed. You can see the input history by executing the '%*history*' command in a code cell.

**Notebook on an exploring interpreter**

Create 10 new code cells (use the 'person' options in the example button to load this example) with the following code snippets:

1. Fact person
2. +person(Alice).
3. -person(Alice).
4. +person(Bob).
5. -person(Bob).
6. +person(Charly).
7. -person(Charly).
8. ?person(Alice).
9. ?person(Bob).
10. ?person(Charly).

Create three different explorations with the following steps:

1. Execute cell 1
2. Execute cells 2 to 7 in any order once
3. Execute cell 8-10
4. Restart by reverting the first node in the execution trace (execution step:1) by pressing 'revert this step'
5. Execute cell 1
6. Execute cells 2 to 7 in any order once
7. Execute cell 8-10
8. Restart by reverting the first node in the execution trace (execution step:1) by pressing 'revert this step'
9. Execute cell 1
10. Execute cells 2 to 7 in any order once
11. Execute cell 8-10
12. Restart by reverting the first node in the execution trace (execution step:1) by pressing 'revert this step'

T1d) Now we have three different explorations. In order to switch to a different exploration, you can make use of the switch button in the 'head nodes' tab. The 'head nodes' tab automatically keeps track of your last explorations, which is sorted by recently updated. Repeat the following tasks until you can not recover an earlier exploration or the number of repetitions exceeds three:

- Switch to the first exploration and execute an additional code cell from cells 2-7 to expand the exploration.
- Switch to the second exploration and execute an additional code cell from cells 2-7 to expand the exploration.
- Switch to the third exploration and execute an additional code cell from cells 2-7 to expand the exploration.

Do the following tasks if possible:

- T1e) Compare the three different explorations and switch to the exploration that contains the most amount of people. Query some of it using cells 8-10
- T1f) Check every exploration that you have attempted and try to recall effects of some operation that you have executed.

**Questions**

Please answer the following questions:

- Q1a: Between a normal notebook and a notebook on top of an exploring interpreter, which one makes it faster to recover a past exploration and why?
- Q1b: Between a normal notebook and a notebook on top of an exploring interpreter, which one took less effort to recover a past exploration and why?
- Q1c: Was there any difference in experience with comparing the explorations between JupyterLab

and a notebook on an exploring interpreter and why?
- Q1d: Between a normal notebook and a notebook on top of an exploring interpreter, what was the difference in experience when checking every exploration and recalling the effect of each operation and what caused this?

### A.2.2 Experiment 2

**JupyterLab**

Create 6 new code cells (press b while not in a code cell) with the following code snippets:

1. x = 111
2. x += 13
3. x -= 17
4. print(x)
5. x *- 7
6. x /= 9

Create 1 exploration:

1. Restart kernel in kernel tab
2. Execute cell 1
3. Execute cell 2
4. Change cell 2 to 'x *=19'
5. Execute cell 2
6. Execute cell 2
7. Execute cell 3
8. Execute cell 4

Freely try out ways to undo/revert one or more previous steps (restarting kernel or making use of checkpoints) in order to execute different code snippets from cell 5 or 6 before printing the end value of x. You are also allowed to create your own code cells if desired.

Do the following tasks if possible:

- T2a) Compare the different explorations that you have experimented with and switch to the exploration with the highest value of x and print it.
- T2b) Check every exploration that you have attempted and try to recall effect of some operation that you have executed. You can see the input history by executing the '%*history*' command in a code cell.

**Notebook on an exploring interpreter**

Create 6 new code cells with the following code snippets:

1. Fact person
2. +person(Alice).
3. -person(Alice).
4. ?person(Alice).
5. ?person(Bob).
6. ?person(Charlie).

Create 1 exploration:

1. Restart kernel in kernel tab
2. Execute cell 1
3. Execute cell 2
4. Change cell 2 to '+person(Bob).'
5. Execute cell 2
6. Change cell 2 to '+person(Charlie).'
7. Execute cell 2
8. Execute cell 3

9. Execute cell 4
10. Execute cell 5
11. Execute cell 6

Freely try out ways to undo/revert one or more previous steps (reverting buttons in the execution trace and in the code cells or making use of tagging nodes) in order to execute different code snippets before querying the facts. You are also allowed to create your own code cells if desired.

Do the following tasks if possible:

- T2c) Compare the different explorations that you have experimented with and switch to the exploration that contains the most amount of people. Query some of it.
- T2d) Check every exploration that you have attempted and try to recall effect of some operation that you have executed.

**Questions**

Please answer the following questions:

- Q2a: What was the difference in experience with the undo/revert functionality between JupyterLab and a notebook on an exploring interpreter?
- Q2b: Where there any functionality of the notebook on top on an exploring interpreter that caused this difference and why?
- Q2c: Was there any difference in experience with comparing the explorations between JupyterLab and a notebook on an exploring interpreter and why?
- Q2d: Between a normal notebook and a notebook on top of an exploring interpreter, what was the difference in experience when checking every exploration and recalling the effect of each operation and what caused this?

### A.2.3 Experiment 3

**JupyterLab**

Freely experiment and prototype different ideas of your choice. Try to use the notebook as an exploratory programming tool. Recover old explorations or use past explorations for new ideas. Write documentation in the markdown cells when appropriate to note your thought process.

**Notebook on an exploring interpreter**

Freely experiment and prototype different ideas of your choice. Try to use the notebook as an exploratory programming tool. Recover old explorations or use past explorations for new ideas. Add annotations in the notebook states when appropriate to write down your thought process. There are examples available in the example tab that you can view and experiment with.

**General questions**

Use your experience of all the three experiments to answer the following questions:

- Q3a: Were there any functionalities of the notebook on top of an exploring interpreter that you used a lot that was useful during exploratory programming that were missing in JupyterLab or vice versa, and why?
- Q3b: Were there any functionalities of the notebook on top of an exploring interpreter that you did not use a lot or were not useful during exploratory programming and why?
- Q3c: What are your overall thoughts about the notebook on top of an exploring interpreter in the context of exploratory programming compared to JupyterLab?
- Q3d: Do you have any suggestions or feedback on how we can better support exploratory programming on JupyterLab or the notebook on top of an exploring interpreter?

## A.3 Implemented functionality of the notebook on top of an exploring interpreter

### A.3.1 Left column

The left column contains the cells (code or documentation) of the notebook and can be interacted with in many different ways with the current state of the notebook.

**Cell actions when pressing 'Actions in the cell'**

**Execute cell** Execute code snippet on the current state of the programming environment.

**Revert to the previous state** Reverts the state of the notebook to the previous state of the executed code cell.

**Revert and execute all** Reverts the state of the notebook to the previous state of the executed code cell and execute all the cells below.

**Swap cell type** Change cell type from code to documentation or vice versa.

**Change cell version** Change cell version to past executed code snippets and their previous/resulting states of the notebook.

### A.3.2 Middle column

The middle column contains the execution trace of the current state of the notebook. The execution trace is represented by the input history of the user and shows each code snippet and its resulting effect (except declarations at the moment in eFLINT) on the notebook.

**Execution trace node actions**

**Convert** Inserts a code cell in the left column with the code snippet that creates the transition from the previous state to the selected state of the notebook.

**Revert this step** Revert to the previous state of the selected notebook state in the execution trace.

**Annotate node** Document the selected state of the notebook in the execution trace.

**Tag node** Tag the selected state of the notebook which saves the state in the tagged nodes list in the right column.

**Execution trace actions**

**Convert to code cells** Converts every node in the execution trace to code cells that can be used to rerun the exploration.

**Inspect trace** Inspect the current state of the notebook and its information.

### A.3.3 Right column

The right column consists of all the exploration that a user has explored. A programmer is able to interact with these explorations by using the revert/inspect/compare commands.

**Switch to tagged nodes/head nodes** Switch between two different tabs that keeps track of the head nodes (automatically) and tagged nodes (manually).

**Switch trace** Switch to the selected state of the notebook

**Remove/hide node** Removes the selected node from the list.

**Inspect trace** Inspects the selected state of the notebook.

**Compare nodes** Compare the selected nodes in the right column list.

# Appendix B

# Survey results

## B.1 Participant 1

**Q0a**
no (not jupyter but similar things like rstudio, matlab, mathematica)
**Q0b**
yes some
**T1a**
yes
**T1b**
yes, but to be able to restore three distinct checkpoints at will, it seemed like I had to make three distinct notebooks (or the save button would just overwrite my checkpoint). Once I had three side by side I could switch tabs to compare their outputs.
**T1c**
yes, the printed history allowed me to recreate any trace I liked in a new cell from start to finish and see all outputs in sequence. It's not very handy but it works.
**T1d**
yes
**T1e**
yes, I could see the sequence of query results right away when switching traces.
**T1f**
yes, the fact that each trace is stack-like, showing results at each step means I can just read over it.
**Q1a**
The eflint one was much faster because every action adds steps onto the stack, so I could decide where to jump to AFTER I did the actions. This is different from jupyter which annihilates previous outputs and only keeps the inputs in the history which I must then copy-paste and rerun (which further complicates the history).
**Q1b**
Same answer as q1b since for me less effort = faster (the notebooks are both fast but me clicking buttons is slow).
**Q1c**
They could both get quite confusing because I still had to model the reasoner's state in my head, but at least in the case of eflint, I had confidence that every exploration really did take me back to SOME correct previous state (where in Jupyter I had to either create loads of checkpoints, or overwrite the state carefully).
**Q1d**
The JupyterLab printed history function was not very handy but it did work when recreating traces. On the other hand, it was really easy with the execution trace in the computational notebook on the exploring interpreter because you could easily read over each node in the trace.
**T2a**
yes

**T2b**
yes
**T2c**
yes
**T2d**
yes
**Q2a**
The eflint notebook tracks every single action ever and makes it possible to revert to that moment. Reverts are also non-destructive, because it creates a NEW sequence of actions, while still keeping the OLD. Jupyter's checkpoint system is very limited because (a) I have to explicitly do it myself, (b) its very hard not to overwrite a previous checkpoint, and (c) when I work hard to have multiple checkpoints, the relationship between these checkpoints is not obvious. Both notebooks offer re-running a previous sequence of steps to recreate some previous state, but in both languages it sucks and it's awkward. The difference is that in Jupyter I am forced to use that, while in eflint, the revert feature does the same job better.
**Q2b**
The biggest contributing factor to eflint being better at this is the fact that its version of checkpoints (nodes) don't overwrite each other, and the sequence between them is tracked explicitly.
**Q2c**
Comparing explorations was actually quite similar. The biggest difference is that since it was less messy to make checkpoints in eflint, it was easier to FIND the outputs of different explorations than it was with Jupyter.
**Q2d**
This was much better with eflint since running the same action again creates a NEW node+output, while running the same Jupyter cell again REPLACES any existing output (if you want to compare both, you need to add new files side-by-side, or recreate previous steps in new cells).
**Q3a**
Exploratory programming with Jupyter was very difficult because the only way to understand the FULL trace in a particular state was to use the history, which kinda defeats the purpose of using a notebook. I missed the node list.
**Q3b**
I only noticed the "Compare Nodes" button at the end. It is very nice because it explicitly lays out the states of different head nodes. The only thing I wished I could do was compare the states of any arbitrary nodes (those before the head) without reverting.
**Q3c**
If you want to do exploratory programming, I definitely think the nodes + traces approach is a step in the right direction.
**Q3d**
I found it very intuitive to reason about the exploration of the program as a tree of actions/steps. I think I would like this treelike structure to be made super explicit, allowing me to (a) inspect the tree visually, (b) arbitrarily mutate the tree node by node, e.g. attach a new node to any existing node, and (c) compute the state of the program from the root up to an arbitrary selected node, potentially comparing these states side-by-side.

## B.2 Participant 2

**Q0a**
yes
**Q0b**
no
**T1a**
yes
**T1b**
no
**T1c**

no

**T1d**

yes

**T1e**

yes

**T1f**

yes

**Q1a**

When there are only two exploration sessions, timing is identical. When there are more, the exploring interpreter makes it much faster via a button click instead of having to execute the whole session again.

**Q1b**

Exploring interpreter because I do not have to perform explicit save points and I do not need to execute the whole session when switching between multiple sessions.

**Q1c**

Yes, exploring interpreter system was trivial to do. In the normal notebook I had to execute the session multiple times, which requires remembering the whole session.

**Q1d**

notebook on top of exploring interpreter made it trivial to see the effects. With the normal notebook I had to remember the effects myself to be able to compare them, which is impossible with a big session.

**T2a**

no

**T2b**

no

**T2c**

yes

**T2d**

yes

**Q2a**

Jupyter only allows one exploration session and reverting loses the other session. This makes it really difficult to compare different sessions. The exploring interpreter notebook allows many sessions and makes comparing trivial.

**Q2b**

revert not destroying the other exploration session.

**Q2c**

Yes. Jupyter required re-executing the session to see the results, which is error prone. In the exploring interpreter comparing was done easily via the click of a button.

**Q2d**

Jupyter notebook it was impossible because I needed to re-execute previous sessions to do the comparison and when comparing the previous session was not available anymore, making the comparison difficult. The exploring interpreter notebook made it very easy with the trace comparison button.

**Q3a**

switching traces and comparing traces.

**Q3b**

No inspection wasn't used, because the trace view gave all information.

**Q3c**

Jupyter notebook seems really restricted compared to the exploring interpreter notebook and does not make real exploratory programming possible. The exploring interpreter notebook does.

**Q3d**

Make Jupyter run on an exploring interpreter. But no, not really an idea.

## B.3 Participant 3

**Q0a**
yes
**Q0b**
No. I know about it, but I have never interacted with it.
**T1a**
yes
**T1b**
yes
**T1c**
yes
**T1d**
yes
**T1e**
yes
**T1f**
yes
**Q1a**
I think the notebook on top of an exploring interpreter makes easier to recover a past exploration. Although interacting with that interface is complex, the Jupyter notebook interface is more user-friendly
**Q1b**
Considering the required effort to understand the notebook interface on top of an exploring interpreter, I think it takes less effort to use the traditional notebooks. However, this is true for simple tasks. Perhaps for other, more challenging tasks (e.g., data analysis), this can be different because users need to keep track of each of their actions.
**Q1c**
As I mentioned before, I think there is a considerable difference between both interfaces. I understand the potential of the exploring interpreter, but the current UI makes it difficult for end-users to see the benefits of such a bookkeeping algorithm. However, in the notebook on top of an exploring interpreter, it is easier to switch between explorations with one click, which is not possible in traditional notebook where you have to keep track of the cells you want to execute, restart the kernel, and run the cells
**Q1d**
One of the main difference I noticed was that in a traditional notebook, I had to keep track of all the changes in the code and the intermediate results, while in the notebook on top of an exploring interpreter, I had trouble understanding how each tab was connected to each other and how to interact with the elements in each of the columns.
**T2a**
yes
**T2b**
yes
**T2c**
yes
**T2d**
yes
**Q2a**
I think that in a notebook on top of an exploring interpreter, it is easier to execute undo actions than in a traditional notebook because there is a dedicated button for this purpose. While in the traditional notebook, users have to restart the kernel or re-execute cells, which might lead the overall notebook to become inconsistent if users try to run all the cells from top-to-bottom.
**Q2b**
Yes, as I mentioned in the previous question, the dedicated button in notebook on top of an exploring interpreter to revert actions is handy. However, it is a feature that the traditional

notebook misses. Although traditional notebook offers workarounds to achieve the same results, I think the approach provided by notebook on top of an exploring interpreter is more straightforward to use despite the UI design.

**Q2c**

Yes, the undo option in notebook on top of an exploring interpreter allows me to compare different explorations easily. Instead, traditional notebook force me to keep track of the alternatives either in my head or by writing them somewhere else (e.g., piece of paper).

**Q2d**

I find challenging to answer this question. On the one hand, the notebook on top of an exploring interpreter allows me to check every exploration quickly. However, recalling the effect of each operation was difficult because there are no explicit hints about how the elements in each of the three tabs relate to each other. On the other hand, in traditional notebook, it is harder to achieve this task, because essentially one has to run and restart the kernel several times to get different explorations, but the results and how each result is related to each other is precise (assuming a re-arrangement of cells to guarantee a linear execution). Moreover, in the notebook on top of an exploring interpreter, I miss the option of writing explanations of my explorations.

**Q3a**

In the notebook on top of an exploring interpreter, when running all the cells, when there is an error, the system still runs all the cells, while in traditional notebook, if there is an error, the execution stops at the cell where the error was found. Another feature that I used a lot in the notebook on top of an exploring interpreter was switching between explorations because it is easy to work on several explorations simultaneously without redoing everything.

**Q3b**

I think there are several functionalities that I did not try (e.g., switch to tagged nodes, inspect nodes), mainly because of the lack of knowledge with the tool, and the experiments were simple, so it was not necessary to use them.

**Q3c**

I think notebook on top of an exploring interpreter offers interesting features that might be valuable for exploratory programming tasks. However, exploratory programming is a complex field, mainly because it depends on each domain and user's background. Moreover, some experience using traditional notebook might affect how users perceive an notebook on top of an exploring interpreter. Therefore, for creating and comparing different explorations, I think notebook on top of an exploring interpreter does a better job than notebook on top of an exploring interpreter

**Q3d**

One of the critical aspects that I missed in the notebook on top of an exploring interpreter is support for documentation. Also, I think the underlying technology of the notebook on top of an exploring interpreter can be integrated within Jupyter notebooks. This might help users identify the benefits of having an exploring interpreter behind a notebook and improve the look and feel of the notebook on top of an exploring interpreter. Finally, I think it would be nice to "tag" explorations with a meaningful name in the head nodes tab.

## B.4 Participant 4

**Q0a**

yes

**Q0b**

no

**T1a**

yes

**T1b**

yes

**T1c**

yes

**T1d**

yes

**T1e**

yes

**T1f**

yes

**Q1a**

Notebook on top of an exploring interpreter

**Q1b**

Notebook on top of an exploring interpreter

**Q1c**

Yes. The exploration in a normal notebook is difficult and prone to be lost. Whereas the on notebook on top of an exploring interpreter supports an easy navigation among the different explorations.

**Q1d**

On the one hand, the normal notebook seems to not be designed to support this user case. Finding the checkpoints and reverting them is an error-prone task. On the other hand, the notebook on top of an exploring interpreter supports directly the case study and the interaction feels intuitive.

**T2a**

yes

**T2b**

yes

**T2c**

yes

**T2d**

yes

**Q2a**

The undo/revert functionality was better supported in the environment with the exploring interpreter. It was cumbersome to explore this functionality in JupyterLab.

**Q2b**

It feels like the environment with the exploring interpreter was designed to support this functionality: having the three different views and the possibility to navigate different explorations made the whole experience smoother.

**Q2c**

Yes. It was very difficult to have a complete image of the explorations when working with JupyterLab.

**Q2d**

It was easier to understand the effects of each operation when working with the environment with the exploring interpreter. You could easily pinpoint what action impacted the data.

**Q3a**

JupyterLab misses the two additional views created in the environment with the exploring interpreter. Being able to switch among explorations and visualizing the order and output of each operation helps tremendously in the exploratory process. In general, it saves time when trying to understand the output of a given execution.

**Q3b**

It feels like the main narrative of the notebook migrates to the execution trace view. The notebook is used as an environment to write any code and all the narrative is kept in the exploration trace. In addition, I did not explore the other options displayed in the head nodes view (e.g. inspect node).

**Q3c**

I feel that the environment with the exploring interpreter fits the needs to perform exploratory programming. JupyterLab is more focused on keeping a linear narrative while the exploring interpreter environment allows the user to create different sequences and readings of the same code.

**Q3d**

I think that the support of the exploratory programming is well supported on the environment with the exploring interpreter. However, I feel that this can be added to JupyterLab in order to have a complete and mature environment that draws on the capabilities and nice user experience of JupyterLab. Furthermore, I think that the notebook as an artifact is getting less attention in

the environment with the exploring interpreter. You can easily use just one cell to create the whole narrative (read in the execution trace view). It would be nice to connect these two views in a such a way that the notebook still has an important role within the exploration.

## B.5 Participant 5

**Q0a**
yes
**Q0b**
yes
**T1a**
yes
**T1b**
yes
**T1c**
yes
**T1d**
yes
**T1e**
yes
**T1f**
yes
**Q1a** & **Q1b**
With an exploring interpreter, it is easier and faster to recover a past exploration. It has a more clear interface showing the past execution and provides buttons to revert to the desired step. In the Jupyter notebook it is more difficult to do the same thing because we can only mark the revert place after one complete exploration. Then we have to repeat the experiment to mark the revert point.
**Q1c**
x
**Q1d**
As for checking exploration and recalling the effect of operations, the interpreter helps a lot on storing the explorations, making it easier to track and trace the different explorations. The interpreter also helps on recalling operations because we can easily revert to where we like with a simple click. Although we can do the same in Jupyter if we have the revert point set, it is not that clear whether we mark the right revert point because Jupyter does not display the check point in its interface.
**T2a**
yes
**T2b**
yes
**T2c**
yes
**T2d**
yes
**Q2a**
In Jupyter I always need to pay attention to each execution of the code cell because it is not easy to undo the execution. Most of the time I need to reproduce the results by reverting to checkpoint and re-execute some more cells if I did not update the check point.
**Q2b**
The exlporing interpreter makes it easier to do so because I can decide the revert point after I execute all the cells. Since it also provides the execution trace, I can easily select the step I would like to revert to. Besides, the "%history" command shows all executions, including the ones I skip by reverting, so it makes it difficult to recognize the operations I add in a new exploration. In the notebook with exploring interpreter, the explorations are recorded in different nodes and

they can be retrieved by simply clicking the "switch trace". It is really convenient when checking explorations.

**Q2c**

x

**Q2d**

In addition, checking the effect of each operation is also easier when using a notebook with exploring interpreter because the output of each step is displayed in the trace. Instead, in the Jupyter I always need to execute print(x) to check the value of x after I make changes to it.

**Q3a**

Exploring interpreter: I like the tag function and the function to inspect trace. They are useful for debugging and to verify the traces. I also like that in the middle column it shows the result of the execution of the cell so I can easily see the effect of them.

Jupyter: The copy and paste functions are sometimes useful to quickly create a small test during an exploration. The markdown cell is also helpful for adding annotations besides the cell.

**Q3b**

Exploring interpreter: I did not use the switch trace a lot and I think the reasone may be that the example case was not too complicate. The query result and effects of execution can be seen easily in the middle column so I don't really need to make many different traces to compare the difference. Change cell version was not used a lot either because I don't have to make too much changes for the code; I can just reuse the cell to produce different outputs.

Jupyter:I don't save checkpoints a lot to revert during the experiment mostly because it is not so easy to confirm where the point is.

**Q3c**

Overall saying the notebook on top of an exploring interpreter provides many features for trace recording and comparing that Jupyter does not have. I can imagine how they can help for exploratory programming when the codes have a proper complexity or when one starts learning programming. Jupyter is more suitable for people who have some experiences of programming and thus have a clear picture of their code traces.

**Q3d**

I believe it can support more if Jupyter add feedback on its checkpoint: maybe to revert the number of steps together or to put a symbol next to the reverted cell. That will make the checkpoint clear to see and make it easier to use. As for the notebook on top of an exploring interpreter, I like many functions it provides for exploratory programming. One thing may in flexibility may be to have the merge/split cell function. For example, once the uncertain cells are checked and verified, they can be merged so the user won't need to scroll a lot to modify the other cells.

## B.6 Participant 6

**Q0a**

no

**Q0b**

yes

**T1a**

yes

**T1b**

yes

**T1c**

yes

**T1d**

yes

**T1e**

yes

**T1f**

yes

**Q1a**

The Notebook on top of an exploring interpreter makes it faster to recover a past exploration because of the head nodes tab and 'revert this step' makes it easy to do so.

**Q1b**

For me the notebook on top of an exploring interpreter took a bit more time just because I confused the switch to tagged nodes and switch trace which make me think that there was something wrong with my browser so I decided to try it with a different browser. In conclusion, it took me more time because of my own fault . If it wasn't for that I would say that the note book on top of an exploring interpreter takes less effort.

**Q1c**

Yes, there was a difference. To switch back to a previous exploration with Jupyter, I had to recall the order of execution (unless there was another way), also repeating the steps was tedious.

**Q1d**

I am not sure, with Jupiter you can see the executions with history, and on the elfint one, you can see that as well. Reverting states was useful also. Making changes was easier not so much retaining the changes made. It was easy to make changes and to see the effect of the changes . Changing persons names, adding or deleting persons was also easier .

**T2a**

yes

**T2b**

no

**T2c**

yes

**T2d**

yes

**Q2a**

Reverting states was useful. Making changes was easier not so much retaining the changes made, on Jupiter I can't recall the first exploration before I made changes to x unless I made the changes to the original again

**Q2b**

I think I answered this for the previous question

**Q2c**

Jupyterlab makes it difficult for explorations because recalling the order of execution makes it tedious

**Q2d**

It was easier to see the effects of the changes you made with then notebook on top of an exploring interpreter

**Q3a**

I am new to the notebook, I can't comment much on that but I do find the notebook on top of the exploring interpreter really useful. It was easy to use and fairly user friendly.

**Q3b**

There were some but I think if I did more experiment on it I would have found them useful

**Q3c**

Its difficult to compare it to something I haven't used much. I found the notebook on top of an exploring interpreter useful. I think it looks super cool as well.

**Q3d**

May be some clarity in the naming of buttons