Exploring Efficient Storage of State in Exploratory Programming

Olaf Erkemeij
o.a.erkemeij@gmail.com

June 18, 2025, 52 pages

Academic supervisor: Dr. L. Thomas van Binsbergen
l.t.vanbinsbergen@uva.nl

Daily supervisor: Damian Frolich MSc
d.frolich@uva.nl

Host organisation/Research group: Complex Cyber Infrastructure (CCI)
Informatics Institute
University of Amsterdam
https://cci-research.nl/

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

Abstract

Exploratory programming is an interactive and incremental style of programming that allows users to iteratively develop programs and compare versions of these programs. This is done by keeping a history of all computational states, allowing users to jump back and forth through states of their program, like traversing a tree. While beneficial for experimentation, storing numerous, potentially large, program states incurs significant memory overhead. This thesis investigates methods to optimise state storage efficiency within an existing generic Haskell back-end for exploratory programming. Through literature review, implementation, and empirical evaluation, six new versions of this framework were developed and benchmarked against the original implementation.

Our primary contribution is a version containing persistent on-disk storage of state, utilising SQLite, JSON patches, binary serialisation, Zstandard compression, that reduces memory usage up to a factor of 26 times, while also enabling session persistence. An in-memory JSON patching approach led to a version of the explorer that has slightly worse or significantly better memory usage, depending on the object language. However, both of these versions incurred significant run-time overhead, leading to at most a 2000x slowdown when executing many small operations. Despite this, for computationally intensive real-world scenarios, the overhead of the persistent storage version is only 17%. This makes it a promising version of the generic exploratory programming back-end, especially for use in fields like data science.

Contents

Chapter 1

Introduction

In 2021, the authors of [1] worked on implementing a framework for exploratory programming. This framework is built in Haskell, to support exploratory programming for incremental programming languages. Incremental means that programs are deterministic and sequential, i.e. the result of executing the whole program is the same as executing its parts sequentially. Exploratory programming is a method of programming, used when both the goal and the path to the goal are not known yet. For example, when working on a dataset, exploratory programming can be used to analyse this dataset.

Exploratory programming is highly relevant within fields of study such as data science [2]. This style of programming allows users to go back and forth through states of their program, like traversing a tree. This allows them to store partial results of their program, and revisit them if needed. Incremental programming tools such as Jupyter Notebooks [1] allow users to write code in multiple cells and use them to build the functionality of their program incrementally. However, a drawback of systems like this is that there is only one global state of the program. When you define a variable and later change the value of this variable, it is no longer possible to go back to its original definition without rerunning the cell in which it was defined. If this cell takes a long time to run, this can cause a lot of overhead. Exploratory programming prevents this by storing multiple states of the program. This means that the user can simply load the state of the program as it was just after defining the variable and continue working with it.

1.1 Problem statement

While exploratory programming is highly convenient for iterative computation and program modification, this form of programming can lead to significant memory overhead. At each point in time, the state, or configuration, of the program needs to be stored in memory, so that it can be reused and revisited at a later point in time. The existing framework stores each state of the program independently of each other. This can lead to substantial memory overhead if the individual configurations are very large (e.g. containing a large dataset), or if the amount of configurations that need to be stored becomes increasingly large. This overhead can degrade performance and make the framework impractical for real-world use.

As the original framework is written in Haskell, it is already able to mitigate part of this limitation through the use of structural sharing. This allows for identical, immutable subcomponents of a data structure, such as the program configurations, to be efficiently stored in memory without redundancy. This sharing is automatically applied to the objects on the heap. However, if two configurations are semantically similar but constructed or stored in ways that result in distinct heap representations, this sharing cannot be applied by the run-time system. Similarly, the way the existing framework represents the data that needs to be stored can cause structural sharing to be ineffective.

This thesis will investigate methods to optimise the storage of state within this existing Haskell-based exploratory programming framework. The main focus is on reducing the overhead caused by storing the states of a program, so that the framework will be a more attractive option for researchers and other users. This should be done using general optimisations that do not require knowledge about the object language (the language using the framework to support exploratory programming). If assumptions about the object language can drastically reduce the memory overhead, they can be considered to be implemented in the framework.

[1] https://jupyter.org/

The research will start by setting up a performance benchmark for the existing framework. This will serve as a baseline for future comparisons. A literature study will then be conducted to find other suitable methods of storing state, and a manual check will be performed to see if they are suitable for the Haskell framework. By implementing these new methods of storing state in Haskell, the performance benchmark can reveal if any improvements have been made. Finally, any drawbacks that occur due to these new methods will be evaluated.

1.1.1 Research questions

To investigate this problem, the following research question will be answered: *To what extent can different methods of state storage improve the memory consumption of a Haskell-based exploratory programming framework?* To answer this research question, the following four subquestions must first be investigated:

1. What is the current memory overhead for state storage in the existing exploratory programming framework?
2. What methods exist for efficient storage of states, and how do they relate to the exploratory programming framework?
3. To what extent do assumptions about the object language help to further reduce the memory usage of the framework?
4. How do the implemented state storage methods compare in terms of memory usage and performance trade-offs to the original framework?

1.1.2 Research method

To answer our research questions, we will begin by finding methods to potentially improve the memory usage of the existing framework. These methods are compared with each other, and a selection is made based on how suitable they are deemed for the framework. Suitability is determined based on compatibility with the current framework and the potential impact of the method. Next, these methods will be adapted and implemented in the framework. To identify which methods will be best suited for the framework, a test suite will be designed. This test suite is capable of measuring both the memory and run-time execution time with a variety of input files. In this way, the original framework can be benchmarked first, and implemented methods can be compared against this original benchmark. As there are currently not many object languages implemented within the framework, the test suite is limited in how many use cases it can benchmark. To remedy this, another programming language will be implemented within the framework to be able to better validate our findings. This will be a language with characteristics that differ from the existing implementations to be able to test a wide variety of test cases.

1.2 Contributions

Our research makes the following contributions:

1. Providing a method to benchmark the memory usage and run-time performance of the exploratory programming framework.
2. Identify several methods that can help to increase the memory efficiency of exploratory programming.
3. Increasing the memory efficiency of the exploratory programming framework.

1.3 Outline

In Chapter 2, we describe the background of this thesis. As this research consists of both empirical and theoretical findings, the remainder of this thesis is structured as follows. In Chapter 3, potential methods to reduce memory usage are identified. In Chapter 4, we describe how the found methods can be implemented in the existing framework. In Chapter 5, we describe how the implemented methods will be benchmarked against the original framework. The results are shown in Chapter 6 and discussed in Chapter 7. Chapter 8 contains the work related to this thesis. Finally, we present our concluding remarks in Chapter 9 together with future work.

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, the concept of exploratory programming is presented, as well as the original Haskell framework developed by Frolich and Binsbergen [1].

2.1 Terminology

We define a set of terms that will be used throughout this research:

Original framework	The exploratory framework as introduced by [1].
Object language	A programming language that uses the framework to support exploratory programming.
State / Configuration	A data structure containing the environmental data needed by the interpreter in order to run a program.
(Execution) History	A data structure containing the programs that have been executed and their output, as well as the configuration that is needed by each program.
Node	A single node in the execution history. The content of this node depends on the specific implementation, but most often contains a specific configuration.
Reference	An integer that refers to a specific configuration in the exploratory programming framework. Can be resolved to its configuration through a lookup in the *cmap*.
Patch / Diff / Delta	A structure that represents a list of operations that can be applied to an object to transform it into another object.
Keyframe	A configuration that is stored in its entirety, instead of being stored as a patch relative to its parent.
(De)serialization	Converting a data structure into a format that can be stored or transferred. It is important that the serialised data can be deserialised to once again form the original data structure.
(De)compression	Compressing a data structure to take up less memory. In this research, compression is always implied to be lossless, as decompressing must lead to exactly the original data.
LRU Cache	Least Recently Used Cache. A data structure that is capable of storing a set amount of objects based on when they were most recently accessed. When data needs to be added to a full cache, the least recently used object is removed.
REPL	Read-Eval-Print-Loop.

2.2 Exploratory Programming

Exploratory programming is an interactive and incremental style of programming, similar to incremental programming. To better explain what exactly this means, the definition of incremental programming is first introduced. Then, a comparison can be made for exploratory programming to show how it differs and what its benefits and drawbacks are.

In incremental programming, a REPL is used to write small pieces of code and immediately see their effect on the overall program that is being written. REPLs are often included within the distribution of a language, such as in Python, or in Haskell through the use of *GHCi*. These REPLs allow for the gradual (or incremental) development of a program, through the consecutive execution of smaller programs. By splitting the program up into small segments, it is easy to see what the effect of a single line of code is. This makes REPLs very suitable for experimenting with different approaches, analysing data, or learning how a programming language functions.

A more popular approach to incremental programming is the use of computational notebooks [3]. These notebooks offer some advantages over REPLs, such as allowing for visualisations and explanations in between the sequences of a program. These visualisations and text snippets allow users to better understand both the individual code segments and the overall program [4]. A popular example of such a computational notebook is Jupyter Notebook [5]. Although Jupyter was originally intended for use with the Python language, it now supports numerous other programming languages.

While these notebooks are great for developing, analysing, and understanding the behaviour of a program, they have their limitations. One of these limitations is how the execution flow of the program is always linear: there is a list of programs that is run sequentially to end up with an end result. This means that if somewhere along this sequence of programs a change is made, the programs starting from this change need to be rerun, and the end result might change accordingly. Due to there always only being one path from start to finish, it is not clear how these changes mid-way influence the behaviour and end result.

Exploratory programming is different, as it allows for changes to be made to the sequence of programs, without overwriting the previous results. This turns the execution flow of the programs from a single line to a branching tree. This allows users to experiment with different versions of a program, and compare their results without losing any information. It also allows for users to jump back and forth to these versions, to pick the one that they like the best. Exploratory programming can be seen as an open-ended version of incremental programming, as it is not yet clear what the desired end result of the program is, or how the programmer will get there [6].

Although exploratory programming can offer significant benefits, it also has some limitations. First of all, there is a lot of data that needs to be stored to make the exploratory process possible. At the very least, all the programs in the tree need to be stored. However, this would mean that to get the output of a program, all programs from the root of the tree up to the newly added program need to be executed. This would be very slow and would hinder usability for the user. A more practical approach is to not only store programs, but also their associated outputs and the underlying environment. Using these environments, each program can be run individually, and no longer requires all parent programs to be executed to reconstruct the needed state. While this makes the process much more user-friendly, it comes at the cost of increased memory usage.

Another limitation of exploratory programming is its lack of proper tooling. Tools like Jupyter Notebook can be adapted to be used with almost any programming language, making incremental programming easily achievable. However, for exploratory programming, such a tool is hard to find. This means that the programming languages themselves need to build tooling to facilitate this. This can be done through the form of an exploring interpreter, which is an interpreter that allows for the use of exploratory programming [7].

2.3 Generic back-end for exploratory programming

In an approach to provide generic tooling for exploratory programming, the authors of [1] built a generic Haskell implementation of an exploring interpreter. This generic implementation was built to support a large class of programming languages, so that by integrating with the framework they can easily provide a language-specific exploring interpreter. This exploring interpreter can then be used, for instance, in a REPL or a notebook-like environment. In this section, the design and implementation of this generic implementation is discussed, as well as its benefits and drawbacks.

2.3.1 Exploring interpreter design

The original generic datatype for the exploring interpreter can be found in Listing 1. Here, *p* and *c* are parameters that need to be filled in by the user, which represent the type of programs that will be executed, and the configurations that are needed to do so. The *defInterp* is given by the user and represents the interpreter that is responsible for executing programs. It takes a program and a configuration and produces a new configuration that holds the effects of the executed program. Next, the *config* field represents the current configuration of the explorer, which will be used next. The *currRef* and *genRef* represent the current reference and the reference that will be used next. These references are used to refer to specific configurations. This allows the execution history of the explorer to not explicitly store the configurations, but only a reference to it. This can be seen in the *execEnv*, which is a graph that stores references to configurations as its nodes and programs as its edges. To resolve those references to actual configurations, the *cmap* is used. Lastly, the user can supply two settings to the explorer, namely *sharing* and *backTracking*. The *sharing* setting denotes whether or not duplicate configurations should be given a fresh reference (so stored multiple times) or reuse the existing reference. The other setting is *backTracking*, which denotes whether going back in the exploration history should delete all nodes and edges in between the current node and the node to which the user is returning.

```
data Explorer p c = Explorer { defInterp    :: p -> c -> c
                             , config       :: c
                             , currRef      :: Ref
                             , genRef       :: Ref
                             , cmap         :: IntMap c
                             , execEnv      :: Gr Ref p
                             , sharing      :: Bool
                             , backTracking :: Bool
                             }
```

Listing 1: The original explorer datatype as presented in [1].

When sharing is turned on, the execution environment becomes a graph, as a node may have multiple incoming edges (programs) that have led to the same configuration. Without sharing, the execution environment is a tree, as there is always at most one incoming edge for every node. When backtracking is turned on, all reverts in the execution history are destructive, causing it to more closely resemble a linked list with stack-like operations. For this research, both sharing and backtracking are always turned off, as the core focus is on improving the memory efficiency of the framework without the optimisation of sharing. As such, they will no longer be discussed.

2.3.2 Core operations

At its core, the generic explorer framework supports five operations: creating an explorer, executing a program, jumping within the execution graph, reverting part(s) of the execution graph, and finally displaying the execution graph.

Creating an explorer

To create an instance of the explorer datatype, a user can call the *mkExplorer* function as seen in Listing 2. This requires them to provide an interpreter for their programming language and an initial configuration. The given configuration is stored in the *cmap* and assigned reference number one. A graph is also made, containing a single node that points to the original configuration.

```
mkExplorer :: (p -> c -> c) -> c -> Explorer p c
mkExplorer interpreter conf = Explorer
    { defInterp = interpreter
    , config = conf
    , currRef = 1
    , genRef = 1
    , cmap = IntMap.fromList [(1, conf)]
    , execEnv = mkGraph [(1, 1)] []
    }
```

Listing 2: The original function to make an explorer instance as presented in [1].

Executing a program

To execute a program within the explorer datatype, the *execute* function can be called. This function receives a program, which is passed on to the interpreter along with the current configuration to produce a new configuration. This new configuration is added to the *cmap* using a newly generated reference. Next, this reference is added as a new node in the *execEnv* graph, with the given program as its incoming edge. Lastly, the current configuration, the current reference, and the *genRef* are updated.

Reverting the execution graph

Another operation that the explorer supports is reverting the execution history back to a specified configuration. The configuration is first found in the *cmap* to ensure that it exists. Then, all nodes that are reachable from that specified configuration are deleted from the execution graph. Lastly, the current configuration and the current reference are set to the specified configuration.

Jumping in the execution graph

Jumping can be seen as the non-destructive variant of the revert operation. After checking if the specified configuration exists in the *cmap*, the current configuration and reference are updated, and with that, the operation is completed. This means that jumping in the execution graph changes the explorer's position in the execution graph, but not the graph itself.

2.3.3 Limitations

While the presented generic back-end for exploring interpreters is a great tool to support exploratory programming for programming languages, it has its limitations. It was built primarily as a proof of concept and has not seen much practical use besides the example integrations shown in the original paper [1] and the follow-up paper in which it is applied to Idris [7]. This means that while the implementation is functional, its memory usage may not be optimal. The exact implementation of the Haskell framework is not discussed in great detail, such as why certain data structures were chosen to represent the data. Therefore, it is interesting to see how different design choices for representing the state required by the explorer interpreter could lead to more optimal memory usage performance.

Chapter 3

Methods for efficient storage of state

In this chapter, the findings of our literature study are presented. We will discuss methods for efficient storage of state that were deemed suitable for the framework. The suitability is decided on the basis of the ease of implementation, the expected performance gain, and the impact on the users of the exploratory programming framework. Each of these aspects are discussed for each presented method.

3.1 Immutable data structures and sharing

Immutability is a key property of functional programming languages such as Haskell. Generally, once a value is created, it cannot be changed. As an example, take a list. An operation that adds an element to a list does not actually modify the list, but rather creates a new version of the list that includes the new element [8, 9]. This simplifies reasoning about the state of a Haskell program, as variables do not change unexpectedly.

Immutability can be used for optimisations in the compiler, such as the principle of structural sharing. When a new data structure is created to replace an old version of it, a compiler like GHC can reuse parts of the old data structure that have remained unchanged. For the example of the list, if the new value is placed in front of the existing list, a new list could be created consisting of the new element, which then points to the old list as its tail. The same principle is applied to numerous tree-like data structures, such as *Map* and *IntMap* [8]. This is typically achieved through *path copying*, where only the path from the root to the new or modified element is updated, while the other branches are shared.

Sharing can significantly reduce the memory usage of the explorer framework if the data is structured correctly. For example, if a configuration of a program is created by appending to a previous version of it, structural sharing could help to avoid storing redundant duplicate data. However, this depends on the data structure that the user uses to represent the configurations and on the data structure that the explorer framework uses to store them.

Another principle that is used through GHC that can help reduce the memory usage of the framework is laziness and memoization. By default, expressions are only evaluated when their results are actually needed. This can prevent unnecessary computation and allows for infinite data structures (such as infinite lists). In the configurations for the framework, some components may remain unevaluated, and as such are stored as thunks (unevaluated computations). Moreover, if a thunk is referenced from multiple locations (shared), its evaluation is performed only once; the thunk is then updated with the computed value, which becomes accessible to all its references.

While thunks can be helpful to prevent unnecessary computations that can slow down the run-time, they can also lead to redundant memory usage. For example, if the thunk is very large, but the end result is not, such as when calculating the maximum element in a very large list. To circumvent this, strictness annotations can be used to force evaluation of data structures. Strictness annotations can be helpful to ensure that the configurations are predictable in their memory usage, rather than depending on to what extent the data of the configuration has been fully evaluated.

Despite the benefits of immutability and structural sharing, it has its limitations in the context of the exploratory programming framework. Just because two configurations are semantically similar or even identical does not automatically mean that structural sharing can be fully applied to them. As mentioned before, this depends on how the interpreter provided by the user handles these configurations, and how they are defined. Moreover, the amount of sharing that can be applied to the configurations depends on the way they are stored in the explorer framework.

3.2 In-memory optimisations

Besides efficiently utilising Haskell's internal mechanisms such as immutability and sharing, several specific in-memory optimisations can be made to reduce the memory usage of the exploratory programming framework. This involves careful consideration and selection of data structures for use within the framework, such that the data that needs to be stored is minimised and optimised. The rest of this section discusses how the design of the explorer in the framework can be modified in order to reduce the amount of data that is stored, in order to improve the overall memory usage.

3.2.1 Tree versus graph structure

As seen in Section 2.3, the original explorer design uses a graph structure for the execution history. This makes sense when sharing is turned on, as that means nodes can have multiple incoming and outgoing edges. However, because sharing is not turned on for this research and reverts are non-destructive, any node in the execution history has exactly one incoming edge (other than the root node). This means that the execution history for this specific implementation can be represented as a tree.

When looking at memory efficiency, storing a tree structure in a graph is not ideal. This is because the graph data structure has to store all incoming and outgoing edges for every node, likely utilising some sort of list. However, for a tree data structure, it is known that there is always at most one incoming edge, removing the need to store a list of incoming edges.

By moving away from the graph data structure that is currently implemented in the explorer, and replacing it with a specialised tree data structure, some memory overhead could be saved. This replacement of the data structure should not affect the core operations of the explorer. The only difference is that, instead of adding the data to a graph, it will now be added to a tree. Depending on if a library or a manual implementation of a tree data structure is used, some auxiliary functions like getting all outgoing edges will require some more implementation work, but should also not be affected in terms of their performance.

3.2.2 Minimal tree structure

While a tree structure already stores less information than a graph structure, it is not minimal. On close inspection of the existing framework and the key operations it performs, it becomes clear that even a full tree data structure might store more information than is essential. In a full tree structure, links to parents and to all the children are stored. However, when looking at the core operations performed by the framework, only the backward edges to the parents are truly needed:

execute : A new node is added as a child of the current state.
jump : No edges required, as a target node is provided.
revert : Since reverts only go back up the tree, following the backward edges should reach the target.

This means that no information about the children needs to be stored in the node, leading to less data being stored. The information stored in a node is then the following:

1. *Config*: The configuration for this node.
2. *Parent*: The reference that leads to the parent.
3. *Edge*: The incoming edge for this node, *i.e.*, the program and output that led to this state. Note that this field is empty for the root node, as no programs have yet been executed.

The nodes themselves would be stored inside of a map, that maps references to the corresponding nodes. This allows for an easy lookup of a parent node, by using the reference stored in the *parent* field. To further clarify how the node structure works, an example can be seen in Fig. 3.1. Here, it shows how the execution environment may look like after executing four programs. A jump has been made after the execution of node three, due to which node two has multiple children.

We claim that this structure is minimal in the context of exploratory programming, as removing any of the components results in being unable to perform the core operations efficiently and directly. Technically, the configuration itself could be omitted, as it could be reconstructed by rerunning the program(s) that led to a node. However, this would make an operation such as jumping to a node very inefficient, as the configuration for the new node has to be built from scratch.

By using this minimal structure, the core operations are unaffected. However, some of the auxiliary operations provided by the framework may be affected in terms of run-time performance. As an example: the *toTree* function converts the execution history of the explorer to a *Tree* data structure. This requires

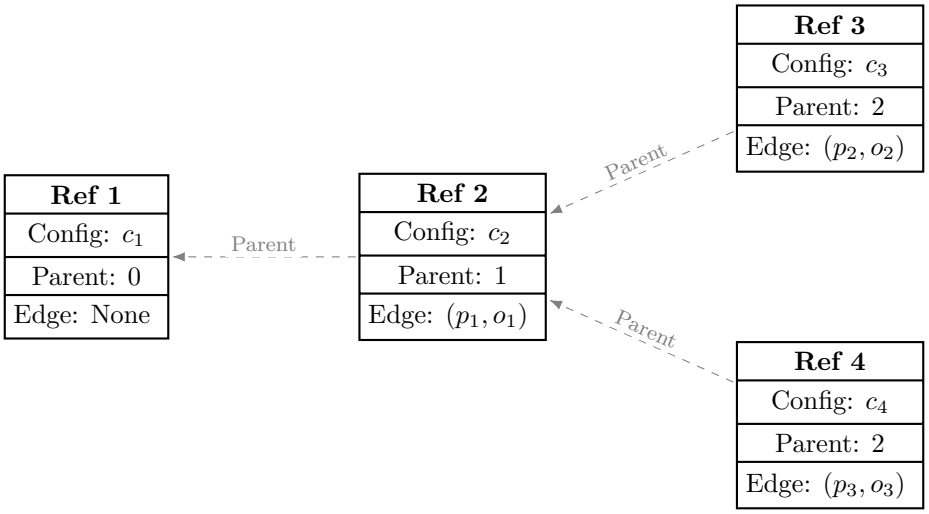


Figure 3.1: Example minimal tree structure for exploratory programming. The edge contains a $(\mathbf{program}_i, \mathbf{output}_i)$ pair, abbreviated to (p_i, o_i).

knowledge of what the children of a specific node are, which requires an $O(N)$ traversal of the map for this minimal structure. As a result, building up a tree goes from being an $O(N)$ operation to being, at worst, an $O(N^2)$ operation. There are ways to mitigate the performance cost of this operation; however, since it is not part of the core features, this is out of scope for this thesis. Nevertheless, it is important to take this into consideration when using the proposed structure.

3.3 data structure patching

While Haskell can perform internal sharing of repetitive data, it is also possible to explicitly add sharing of data to the explorer framework. The core observation here is that, most of the time, configurations stored in the explorer are built on top of each other. When a program is executed, the current configuration is used, and a changed version of this configuration is then stored for the next node in the explorer. While there may have been data added or altered within the configurations, there will likely be parts that have remained unchanged. Instead of storing two full (potentially large) configurations, only storing the changes (a delta) to the configuration would suffice.

This approach draws inspiration from various places. For example, using Git [1], users can view the changes that have been made to a file, and apply them to get to a new version of that file. Deltas are also commonly used within video compression [10]. Here, instead of storing a full video frame, only the pixels that change relative to the previous frame can be stored. Then, by applying those changes, the full frame can be reconstructed using the previous video frame.

To apply this idea to the exploratory programming framework, a technique must be devised to compute and apply deltas given the configurations supplied by the user. Ideally, utilising this technique should come with little or no added complexity for the end user. However, within Haskell, a problem arises when attempting to apply this technique to functions. If a user has functions stored in their configurations, such as for the environment of the program, deltas cannot easily be computed. This is because every computation of a configuration delta requires some sort of serialisation of this configuration. This is a problem, as functions cannot be serialised. This means that the user cannot directly use Haskell functions within their configurations. While this is a limitation, it does not mean that users cannot represent functions within their configuration at all. An example alternative approach to representing functions within the configuration could be through an AST. This is a type of language design more formally known as a deep embedding of a domain specific language [11].

3.3.1 Diff types

Multiple options to compute deltas of Haskell data types exist, but they vary in terms of their performance and their ease of use. In this subsection, three potential methods are discussed and compared.

[1] https://git-scm.com/

Structured

Ideally, the patches created by the diffing algorithm should be as compact as possible, to maximise the amount of data saved. The more knowledge the underlying algorithm has about the semantics of the data for which a patch is being computed, the more efficient the resulting patch will be [12].

An example of such a tool written in Haskell is *gdiff*, based on the work of Lempsink [13]. This library claims to efficiently produce optimal patches for datatypes in Haskell, which would be ideal for the exploratory programming framework. However, the library requires the definition of several class instances for the datatypes on which the delta is computed. This means that if a user wishes to add his programming language to the exploratory programming framework, this requires them to define these instances themselves. This requires extra effort, as users will need to learn how this library works and apply it to their code.

In conclusion, while this technique for computing deltas is efficient and optimal, it adds a significant burden to the users of the framework.

Binary

Another possibility is to first convert the data to binary, and compute deltas on that. There are several tools that compute binary deltas, one of the most performant being *bsdiff* [14]. The benefit of using an algorithm such as this is that converting Haskell datatypes to binary data is derivable. This means that adding support for conversion to binary is straightforward, as it can be done by deriving a generic instance of the *Binary* typeclass. In addition to that, users of the exploratory programming framework do not need to know how the explorer generates and uses deltas.

The disadvantage of using *bsdiff* is that it requires a lot of memory to compute deltas [14]. This means that while we are lowering the average memory usage of the explorer, we might be raising the peak memory usage that the explorer uses. Furthermore, there is no direct implementation of *bsdiff* in Haskell, which means that an external executable to compute the deltas would become part of the explorer.

In conclusion, while this technique is easier to use for users, it is not the easiest method to implement within the explorer and comes at the cost of increasing the peak memory usage.

JSON

Finally, as an alternative to converting to binary, JSON data may be considered. There is an official JSON document structure that describes a series of operations that are able to convert one JSON document to another [15]. This method has the same ease of use benefit as the binary approach, as *ToJSON* and *FromJSON* are two typeclasses that can be derived in Haskell.

Moreover, a Haskell library, *aeson-diff* [2] has been developed that adheres to the JSON patch structure to compute deltas between JSON structures and apply them. This means that implementing this technique into the explorer framework can be performed quite easily. However, just as is the case with the binary approach, the deltas that are computed might not be optimal for the original data structures. Despite that, it will still be efficient as it follows the official patch structure for JSON documents.

In conclusion, converting datatypes to JSON to compute and apply deltas is an efficient approach that offers competitive performance and great ease of use for both the developers of the exploratory programming framework and its users.

3.3.2 Keyframes and checkpoint strategies

While storing deltas instead of full configurations can lead to optimised memory usage, it comes with some challenges with regard to the execution time. To reconstruct a specific configuration using the stored deltas, a source configuration is needed. This source needs to be stored in full, and is what has been used to compute the delta, and what the delta needs to be applied to end up at the target. From this point on, these source configurations will be referred to as a keyframe, inspired by how this works in video compression [10].

The main challenge lies in the fact that the further away a configuration is from a keyframe, the more deltas need to be applied to reconstruct that configuration. This means that the larger this distance gets, the more time it will take to reconstruct. In the most simple application of storing and applying deltas, there is only one keyframe: the root of the exploration tree. To reconstruct the configuration of

[2] https://hackage.haskell.org/package/aeson-diff

a specific node in the tree, the path from the root to that node must be walked. The length of this path is, in the worst case, equal to the depth of the tree. This, in turn, can in the worst case be $O(N)$, when the tree models a linear execution graph (no jumps have been taken). While this overhead might be acceptable for a small number of states, this can significantly slow down the explorer the larger it gets.

To remedy this performance bottleneck, a checkpoint strategy needs to be chosen: What nodes will be saved as keyframes? Only saving the full configuration as a keyframe might be the most memory-efficient option, but also the least performant in execution time. On the other end of the spectrum, saving every node as a keyframe would be the most performant in execution time, but least memory-efficient. So, a checkpoint strategy needs to be somewhere on this spectrum, depending on which end has the highest priority.

A simple solution would be to save a node as a keyframe every N nodes. This ensures that the reconstruction paths will always be at most N steps long. Another strategy could be to implement a probabilistic checkpointing system: Let there be a chance p that a newly added node becomes a keyframe. However, adaptive strategies could offer a better balance. For instance, deciding to turn a node into a keyframe when it has reached a certain outdegree, or after a jump has been performed. The size of the patch could also be considered in whether or not to store a configuration fully or not. In short, choosing a good checkpoint strategy can determine the effectiveness of this technique.

3.4 Compression algorithms

Another idea for reducing the amount of data that needs to be stored is by compressing it. A compression algorithm could make the configurations themselves smaller, leading to reduced memory usage. Then, only when the data is needed will it be decompressed. To make this technique effective, a good compression algorithm needs to be used. First of all, the algorithm needs to be lossless, as the decompressed data must be identical to the original data. In addition, the algorithm must offer a good balance between the compression ratio and the speed of compression and decompression. If the compression ratio is too low, compressing the data makes little sense. But if the (de)compressing time is too high, compressing the data no longer becomes worthwhile within the context of exploratory programming.

One of the most widely adopted and used compression algorithms is *zlib* [16]. It offers a high compression ratio and generally good compression speed. However, a more recent compression algorithm is *zstd* [17]. This algorithm offers a good balance between its compression ratio and speed. Compared to *zlib*, the compression ratio is slightly behind, but it has higher speeds [18]. Due to this good balance in speed and ratio, *zstd* is deemed the best choice for use within the explorer framework.

3.5 Persistent storage

Another possible approach to reduce the memory usage of the framework is by storing the relevant data on disk. This way, the same amount of data is still stored, but it now uses storage space on the host computer instead of memory within the application. This approach offers multiple potential benefits, as it allows for the use of large histories that would not have fit in memory, and it allows for sessions to be stored and continued. Moreover, data might be able to be stored more efficiently, using techniques that would not be of use within the Haskell application.

In Haskell, data can be exported to the disk through the use of serialisation and IO. By first converting the data to a structure such as plain text, bytestrings, or JSON, this can then be written to the disk. To reconstruct the original data, it needs to be read from where it is stored and then converted back into the original data structure. This could be applied, for instance, to the configurations stored in the *cmap* of the explorer, or to the nodes of the minimal tree structure as discussed earlier.

In terms of how exactly to store the data, there are two main options. First, the relevant data could be stored in a custom file format. The benefit is that this means the data can be stored in any format, as the framework itself is responsible for parsing the data back into the original data structures. For example, configurations could be stored in bytes, and each line in the storage file represents one configuration. This way, the line numbers of the file can be used to number the configurations, allowing them to be looked up by their reference number. A drawback of this approach is that the implementation work required to ensure the data is stored and reconstructed correctly is significant.

Another option is to use a database to store the data. A benefit of using a database is that a lot of implementation work is already present, such as efficiently storing and retrieving the data. In terms of choosing a database, there are also two options. An option is to use a standalone database that runs

in a container such as Docker. This allows for the use of almost any database but comes at the cost of more complex usage and more dependencies. It requires the users of the exploratory programming framework to run such a container themselves, which adds significant complexity. Another option is to use an embedded database, which is a database that is tightly integrated with a programming language such as Haskell. Within Haskell, an example of this is *SQLite* [19]. This database can be interacted with through the use of a Haskell library, making it easy to use. It also does not add extra complexity or dependencies to the end-user. *SQLite* is known to be a database with little overhead, making it suitable for the use of local data storage [19].

While storing parts of the data for the explorer framework on disk can offer memory usage benefits, it comes at the cost of increased execution time. Interacting with data stored on disk is inherently slower than data kept in memory, so a strategy is needed to minimise the amount of disk IO. When using a database to store the data, this is already done to some extent. For example, the database keeps a cache for recently queried data. However, an in-memory cache can be added to the explorer framework to further enhance the performance. This in-memory cache can keep data that has been retrieved from disk, so that when it is needed again the disk IO can be prevented.

The approach for persistent storage is particularly promising when combining it with other techniques for optimisations. For example, instead of full configurations, the deltas could be stored. These deltas could also be compressed, for example, by using *zlib*. Through this hybrid approach, memory usage could be greatly improved, at the cost of some run-time performance.

Chapter 4

Implementation

Having found the methods that can make the exploratory programming framework more efficient in terms of memory usage, they can now be adapted and implemented. To do so, the methods need to be implemented in the Haskell codebase. Each method that is implemented leads to a different version of the framework, which is numbered so they can be more easily compared later. To this end, the original framework has been labelled version 1. Each version will be presented individually, highlighting important design choices and the changes made compared to the original framework. For version 1, the Haskell-specific details of the framework are discussed, as they differ slightly from the design as presented in Chapter 2.

An important thing to note is that for all versions, strictness has been implemented wherever possible for the data stored in the explorer. The reason for doing so is twofold. For one, it forces Haskell to fully evaluate the data fields, which can be more efficient in terms of memory usage. This also allows for a more fair comparison between versions of the explorer, as it eliminates the possibility of some data being stored lazily in one version but fully evaluated in another.

4.1 Version 1: Original framework

The first version of the framework is the original implementation of the exploratory programming framework. This version uses a graph to store the execution environment of the explorer through the *fgl* [1] library [20]. In this graph, the nodes are references that represent the configurations of the explorer. The edges of the graph consist of the programs and outputs that led to the configurations. The graph is stored in a patricia tree, which allows for efficient storage and retrieval of the data [21]. Besides the graph, the explorer also contains an intmap that maps references to configurations. The full datatype of the explorer can be seen in Listing 3. This implementation differs from the one presented in Section 2.3, mainly in the datatypes for the interpreter and the execution graph. The interpreter now returns a *Maybe* configuration, and the output of the interpreter. This output is explicitly stored in the execution graph, alongside the program. This is done to enhance the reproducibility of the output and to be able to use it in any tooling that requires it [1, 22].

Moreover, the interpreter now takes a new type parameter o for the output, which is restricted to the class of monoidal types so that the output can be concatenated. There is also a new type parameter m, which represents the monad in which the interpreter performs computations. This could allow an interpreter to perform IO actions through use of the IO monad and still work with the generic framework.

The key operations of the explorer are *execute*, *jump*, and *revert*. The *execute* operation executes a program and stores the resulting configuration in the explorer. This is done by adding a new node to the *execEnv* graph and storing the new configuration in the *cmap* intmap. The *jump* operation jumps to a specific node in the explorer and sets the current configuration to that node. The *revert* operations revert to a given node, by first checking if there is a downward path from the target node to the current node. If this is the case, all nodes on this path are removed from the graph, and the current configuration is set to the target node.

The original framework relies on Haskell's internal sharing to reduce the memory usage of the configurations. The expectation is that this version of the framework uses a lot of memory, as it potentially stores a lot of redundant data. Moreover, the graph data structure used to store the execution environment

[1] https://hackage.haskell.org/package/fgl

```haskell
type Ref = Int

type Language p m c o = (Eq p, Eq o, Monad m, Monoid o)

data Explorer programs m configs output where
  Explorer ::
    Language programs m configs output =>
    { defInterp :: programs -> configs -> m (Maybe configs, output),
      config    :: !configs,
      currRef   :: !Ref,
      genRef    :: !Ref,
      cmap      :: !(IntMap.IntMap configs),
      execEnv   :: !(Gr Ref (programs, output))
    } ->
    Explorer programs m configs output
```

Listing 3: The original explorer datatype in Haskell.

has more overhead than needed, which is also expected to lead to increased memory usage.

4.2 Version 2: Explicit tree structure

The first change made to the original framework was to replace the graph representation for the execution environment with a tree representation. This has been implemented through the use of two intmaps:

1. `parents :: IntMap (Ref, (programs, output))`. Used to store the backward edge of a node.
2. `children :: IntMap [(Ref, (programs, output))]`. Used to store the forward edges of a node.

This does not drastically impact the original framework or its operations. Some functionality that the graph representation offered, such as getting all the edges of the graph, had to be replaced by a more explicit version that works with the two intmaps.

This version stores less data for the execution environment, as it is now stored in two intmaps instead of three (see Chapter 2). Because of this, the expectation is that this version of the framework uses slightly less memory than the original.

4.3 Version 3: Configurations vector

The second change made to the framework builds on top of version 2, as it contains the same tree structure. The key change in this version is to swap out the intmap used to store the configurations with a vector: `cmap :: Vector (Maybe configs)`.

The configurations stored in the vector are wrapped in a *Maybe* datatype, as they could have been deleted as part of a *revert* operation. This subsequently means that the size of the vector does not shrink in the case of a *revert*, as the deleted nodes keep their location in the vector. This decision was made because the index of a configuration corresponds with its reference. If the vector were to be compacted by removing the empty entries, the reference of nodes may change. This would require a complex operation to update all uses of the changed references, which would make using a vector not worthwhile.

Other than the *revert* operation, the other functionality does not change much. Some operations that were performed using the intmap capabilities, like *filterWithKey*, had to be replaced with equivalent operations for the Vector datatype.

The expectation for this version is that if little to no reverts are performed on the explorer, the Vector can more compactly store the configurations than an intmap. This should lead to a bit less memory usage than both version 1 and version 2.

4.4 Version 4: In-memory patching

The fourth version of the framework is based on the idea of data structure patching, as discussed in Section 3.3. It is an adaptation of the first version, but instead of the *cmap* directly storing configurations, it

now stores an embedded version, as seen in Listing 4. This new datatype stores either a full configuration (in the case of a keyframe), or a patched configuration. In the case of a patched configuration, it stores a base reference from which the patch data was generated, the patch data itself, and the distance from here to the base reference.

```
data ConfigDiff c = FullConfig !c
                  | PatchedConfig { baseRef   :: !Ref
                                  , patchData :: !Patch
                                  , distance  :: !Int
                                  }

cmap :: IntMap (ConfigDiff configs)
```

Listing 4: The new datatype for patching configurations, and the updated *cmap* that makes use of this new datatype.

Since the *cmap* now no longer directly stores configurations, a mechanism is needed to reconstruct a specific configuration in case it is needed in the framework. This is handled by the *deref* function, which normally resolves a reference to its corresponding configuration. If the reference points to a full configuration, this is simply returned. If the reference points to a patched configuration, the function *getConfigRecursive* is called. This function first looks at the parent reference of the node whose configuration needs to be reconstructed. Using this parent reference, a recursive call is made to first *deref* that reference, and resolve the parent configuration. Using the parent configuration, the stored patch data can be applied to obtain the requested configuration.

Another operation that slightly changes in its behaviour is *execute*. Instead of just adding the new configuration to the *cmap* of the explorer, it now first needs to be turned into a *ConfigDiff*. To do this, a decision must be made whether to store the configuration as a keyframe or not. It should be noted that the root node is always a keyframe. In this version, the checkpoint strategy depends on the *distance* parameter stored in the *ConfigDiff*. If the distance from the newly created node to the last known keyframe is bigger than a threshold d, the new node becomes a keyframe. Otherwise, a patch is created by converting both the new configuration and the current configuration to JSON, and calling the *diff* function from the *aeson-diff* library.

By default, the value for d is set to 5. This means that the length of patches needed to be applied before reconstructing any configuration will never be larger than 5 for the default version.

To avoid having to reconstruct the current configuration for the explorer, the current configuration is stored as a separate parameter within the explorer datatype. This parameter is updated whenever the current configuration changes, so it is always up to date.

The expectation is that by mostly storing patched configurations, the total memory usage of the framework will go down. This will come at the cost of some run-time performance, but since the maximum length of patches is limited, the expectation is that the impact will be small.

4.5 Version 5: Minimal tree structure

For the fifth version of the framework, the *execEnv* and *cmap* of the original framework are replaced with an implementation of the minimal tree structure as discussed in Section 3.2.2. The Haskell implementation of this datatype can be seen in Listing 5.

```
data ExpNode p c o = ExpNode
  { nodeConfig :: !c,
    nodeParent :: !Ref,
    nodeEdge   :: !(Maybe (p, o))
  }

history :: !(IntMap.IntMap (ExpNode programs configs output))
```

Listing 5: The new datatype to represent the minimal tree structure, and the history field that replaces both the *cmap* and *execEnv* of the explorer.

The core operations of the framework do not change much, as data is now simply inserted in the *history* datatype rather than the original fields. One core operation whose behaviour does change is the *revert* operation. This operation now utilises a helper function that tries to find the path from the current configuration to the one that needs to be reverted to. This is done by walking up the tree, until either the destination is found or the root node is reached. The nodes that are on the path are deleted, and the current configuration is set to the target configuration.

One of the auxiliary functions whose behaviour has to change significantly is the *toTree* operation. This operation now works by invoking a helper function *findChildren* for each node, which finds the children for this node. This causes the whole tree to be traversed for every node in the tree, leading to $O(N^2)$ run-time performance. As explained in Section 3.2.2, optimising this function is out of scope for this thesis.

The expectation is that this version of the framework outperforms both version 1 and version 2 of the framework. Where version 2 stores less data than version 1, this new version 5 stores even less data for the history of the explorer. Besides some of the auxiliary functions that require access to the children of specific nodes, we expect that the run-time performance of the operations stays the same. This is because the run-time complexity of the core operations has not been affected by this change.

4.6 Version 6: Configurations compression

The sixth version of the framework is largely similar to version 1, but adds compression to the *cmap*. Instead of storing the configuration itself, it is now stored as a bytestring, as can be seen in Listing 6. All the operations stay the same as in version 1, except that *execute* now encodes and compresses the new configuration before storing it. When retrieving a configuration from the explorer using *deref*, the configuration is decompressed and then decoded before being returned to the user.

```
cmap :: !(IntMap.IntMap ByteString)
```

Listing 6: The changed *cmap* for the sixth version of the framework.

The expectation is that this version of the framework uses less storage for the configurations, and should therefore lead to lower memory usage.

4.7 Version 7: On-disk patching and compression

The seventh and final version of the framework is in large part a combination of all previous versions, along with the notion of persistent storage as discussed in Section 3.5. It uses the minimal tree structure from version 5, along with the patching strategy from version 4. This information is then compressed and stored on disk through the use of a database. The rest of this section goes over each design choice in more detail, along with how the core operations are implemented. It should be noted that besides the *toTree* function, only the core functionality for the explorer is implemented. This is because those operations together form the basis for a proof of concept, as they can be used to design a REPL to work with this explorer.

4.7.1 Database design

The database chosen for this implementation is *SQLite* [19]. This database is deemed the best choice for the explorer framework for a number of reasons. First off, being an embedded database, it can be easily integrated into Haskell through the use of a library package. Second, it is small, fast, and self-contained, meaning it can run on any platform and offers excellent performance [19]. Other databases might provide better run-time performance, but this would come at the cost of increased overhead or decreased compatibility across operating systems.

For this project, the *sqlite-simple* [2] package was chosen to act as the SQLite client library. It is a mid-level client, allowing the writing of direct SQL queries, but also providing some automatic data conversion between database and Haskell types.

To store the data of the explorer, a database file is initialised when the explorer is created. Where the database file is stored is up to the user, as they are required to provide a filepath. In this new database

[2] https://hackage.haskell.org/package/sqlite-simple

file, a table is initialised that will store the relevant information for the explorer. The design of the table can be found in Table 4.1. It mostly corresponds to the minimal tree structure discussed earlier, but contains an extra *checkpoint* field to indicate if the compressed configuration data is a patch or a full configuration.

Column	Type	Constraint	Description
ref	*INTEGER*	Primary key	Reference of the node
parent	*INTEGER*	Not null	Reference of the parent
checkpoint	*BOOLEAN*	Not null	Denotes if this node is a keyframe
config	*BLOB*	-	The configuration data (compressed)
edge	*BLOB*	-	The incoming edge data (compressed)

Table 4.1: Database schema of the *History* table, used to store the explorer data.

When the database is initialised, an index is also added on the *parent* column of the table. While this takes up some more memory, it allows for fast lookups of nodes both by their reference and by their parent field. This means that querying which configurations in the database have a specific node as parent becomes a fast operation, which helps functions such as *toTree*.

To support the operations needed by the explorer, a number of helper functions have been written to interact with the database. These allow users to initialise a database file and connection, close the connection, and delete the database. It also offers support for writing the data of a node to disk, as well as retrieving the data for a specific node. Moreover, it supports deleting a number of nodes and finding the children of a specific node.

4.7.2 Compression technique

Before any data is written to the database, it is first compressed. To this end, the *zstd* compression algorithm has been used, as motivated in Section 3.4. Both the configuration data and the edge data of the nodes are compressed using this algorithm, to minimise the amount of data that needs to be sent and stored in the database.

The compression level that the framework uses is configurable by the end-user. As a default, level 3 is chosen, which is also the recommended default for *zstd* [23]. This level offers good levels of compression ratio and speed, which is a good fit for the explorer framework.

4.7.3 LRU caching

Storing data in a database file as opposed to in-memory can lead to slow access times, as disk IO is required to retrieve the data. To remedy the effect of this on the explorer framework, an LRU cache has been implemented. This cache stores recently accessed configurations so that they do not need to be reconstructed if they are needed again in the near future. This is a critical speedup because it avoids fetching all nodes needed to reconstruct a specific configuration.

By default, the cache contains at most 10 nodes, before it throws out the oldest configuration to make room for new ones. The size of the cache is also configurable by the user.

4.7.4 Checkpoint strategy

To avoid having to store additional information in the database, the checkpoint strategy for this version has been kept simple. Every N_{th} node that is added to the explorer becomes a keyframe node, starting with the root node. So, if $N = 5$, in an explorer of size 12, the following nodes would be keyframes: 1 (the root), 6, and 11. The default value for the checkpoint interval is 5, but can be changed by the user.

4.7.5 Explorer design

Since interacting with a database inherently requires IO, the design of the explorer datatype has to be revisited. Initialising the database returns a connection that can be used to interact with the database. This connection should be kept alive and only closed when no longer needed. It would not make sense for this version of the explorer to be immutable, so instead it is wrapped inside a mutable reference through

the use of *IORef*. This allows the explorer, the database connection, and the LRU cache to be initialised once, and then modified when needed. The design of the explorer data type can be found in Listing 7. Here, *DiskStoreHandles* represents the database connection generated by the helper functions.

```
data ExplorerState p c o = ExplorerState
  { diskStore   :: !DiskStoreHandles,
    cache       :: !(LRU.AtomicLRU Ref (ExpNode p c o)),
    currRef     :: !Ref,
    genRef      :: !Ref,
    interpreter :: p -> c -> IO (Maybe c, o),
    checkpoint  :: !Int,
    compression :: !Int
  }

data ExpNode p c o = ExpNode
  { nodeConfig :: !c,
    nodeParent :: !Ref,
    nodeEdge   :: !(Maybe (p, o))
  }

newtype Explorer p c o = Explorer (IORef (ExplorerState p c o))
```

Listing 7: The changed *Explorer* type for for the seventh version of the framework.

Another thing to note about the design of the explorer is that it can no longer take place within any variable monadic environment, as was the case for the original design. Due to the interaction with the database, the explorer inherently operates within the IO monad. This means that the interpreter supplied by the user must output `IO (Maybe configs, output)`. While this constrains the interpreter to the IO monad, this is often a natural choice for interpreters that may perform IO or manage complex state, and is therefore not considered a major practical limitation for many use cases. Furthermore, if the interpreter output is not in any monad, it can easily be turned into IO through the *return* statement.

Finally, the explorer now also supports continuing working on a previous session. This is done by adding a new initialisation function, *mkExplorerExisting*, which requires a path to the database and the interpreter to use. This function will use this database for the explorer and set the current reference to the last known reference in the database.

4.7.6 Execute implementation

When executing a program, three key things must be stored in the database: The new program, its output, and the new configuration it created. To do this, it must first be decided if the new node will be a keyframe or not. To do so, the following check is performed: $(\text{Ref}_{\text{new}} - \text{Ref}_{\text{initial}}) \mod \text{checkpoint} \equiv 0$. When this is the case, the configuration will be stored in its entirety. Otherwise, the configuration is stored as a patch. This patch is created by retrieving the current configuration, converting both the current and new configuration to JSON, and calling the *diff* function from the *aeson-diff* library.

It can be the case that the new configuration is the same as the current configuration. This results in an empty patch, which on itself takes up some space in the database. To avoid this, the patch is only created if the new configuration is different from the current configuration. If they are the same, the new configuration is set to *Nothing*, indicating that it is the same as its parent. This requires special consideration when reconstructing the configuration, as will be explained further on.

To store the data in the database, the edge and configuration data (or patch) are compressed by encoding them to a strict bytestring, and then compressing them using the *zstd* algorithm. All the relevant information is then sent to a helper function, which writes it to the disk. Then, an *ExpNode* instance is created for the new node, so that it can be inserted into the LRU cache.

Finally, the explorer is updated, by setting the value of the current reference to the reference for the new node, and incrementing the reference generator.

4.7.7 Jump implementation

For the jump operation, the reference of the node to jump to needs to be resolved. This is done through a helper function called *getNodeIO*. This function first checks if the requested node is in the LRU cache, and if so, returns it. If not, the node will have to be reconstructed. To this end, the raw data for the node is retrieved through the database. The edge can immediately be reconstructed, by decompressing the raw edge data and decoding it. The edge data can be *Nothing*, in the case that the node is the root node, or it can be a regular (program, output) tuple.

For the raw configuration data, there are three possibilities. The data can be empty, indicating that the configuration is the same as its parent. In this case, the function recursively tries to reconstruct the parent configuration. The second possibility is that the node is a keyframe, in which case the full configuration can be reconstructed by decompressing the data and decoding it. The third possibility is that the node is a patched configuration. In this case, the parent configuration is reconstructed, and the patch data is decompressed and decoded. The patch is then applied to the parent configuration to form the final configuration. Finally, an *ExpNode* instance is created for the node, and it is inserted into the LRU cache and returned.

4.7.8 Revert implementation

To revert to a specific node, the process that is followed is similar to the one detailed in Section 4.5. The main change is that each node has to be reconstructed through the *getNodeIO* function before it can be used.

4.7.9 ToTree implementation

The *toTree* function also follows the same logic as described in Section 4.5. The *findChildren* is now a helper function that performs a query on the database to find all the children of a specific node. All these children then have to be reconstructed before they can be added to the tree. This means that the function not only still has a $O(N^2)$ run-time complexity, it also has IO overhead for each node. As mentioned before, this limitation is out of scope for this thesis and will not be optimised.

4.7.10 REPL implementation

As the new explorer lives within the IO monad, the traditional REPL implementation of the original explorer cannot be used. To remedy this, a new REPL has been implemented. This REPL behaves identically to the original, but now uses the mutable IO explorer.

4.7.11 Expected benefits and drawbacks

The expectation for this version of the framework is that it will use less memory than all previous versions while also being able to store the data persistently. This means that the explorer can be used over multiple sessions, and the data will not be lost when the program is closed. The run-time performance is expected to be worse than the in-memory versions since IO overhead will be present for every operation. However, the LRU cache should help mitigate some of this overhead.

Another benefit of using persistent storage is that the explorer can be stopped and continued at a later time, which is not possible with the in-memory versions. The expectation is that this extra option will benefit the user-friendliness of the explorer.

Chapter 5

Experimental Setup

To be able to evaluate and compare all implementations of the framework, they must be tested. First, the existing framework should be tested so that it can act as a baseline. Then, the same tests can be repeated for each version of the framework, so they can be directly compared. This chapter goes into what languages are used to test the versions, what exactly will be tested, and how these tests will be performed.

5.1 Object languages

In order to be able to use the explorer, an object language must exist that can integrate with the framework. To this end, an example language is present within the original framework, the Mini-Java language. However, only measuring for one specific language would limit the extent to which the results are generalisable. It might well be that other object languages that are implemented in a different way or have way different characteristics would lead to different results. To remedy this, another object language has been designed to work with the explorer: a variant of the Scheme language.

The rest of this section goes into detail on how each of these languages is designed, what their characteristics are, and how they differ from each other.

Mini-Java The Mini-Java language is designed as a subset of the Java language. It was originally designed as a compiler project to teach compiler implementations in Java [24, 25]. Despite being more than two decades old, the Mini-Java language is still relevant to teaching compiler design, as it is still being used and adapted for modern use cases [26].

It is an object-oriented and imperative programming language. Programs consist of classes, which, in turn, contain methods and variable declarations. In terms of data types, Mini-Java supports integers, booleans, and arrays of integers. The available operations consist of basic arithmetic (addition, multiplication, subtraction), boolean logic (!, &&) and comparisons (<). In arrays, the *length* function is defined to determine the length of an array.

In terms of statements, Mini-Java supports *if* for branching and *while* for loops. Additionally, printing is supported through `System.out.println`, which appends strings to a list that is kept in the program's configuration. Variables and array elements can be assigned values multiple times.

To facilitate user interaction through the use of a *REPL*, the Mini-Java language exposes a *Phrase* datatype. A phrase can be an expression, a statement, a class, a method, or a method call. This allows for the execution of code such as an expression on a global level, which would not be possible in a normal Mini-Java file. This characteristic makes Mini-Java well-suited for using it as an exploring interpreter, as it naturally supports incremental construction and evaluation of programs.

The execution state of a Mini-Java program is modelled by the *Context* data structure, which serves as a configuration. The *Context* is also passed to the exploratory programming framework, which uses it along with the Mini-Java interpreter to provide an exploring interpreter. The complete *Context* data type can be found in Listing 8.

A key element in this data type is the *env* value, which maps identifiers, such as variable names, to their corresponding values. This value can be a literal (boolean or integer), a reference to heap-allocated objects, or a class definition. The heap allocation is modelled through the *store* field, which maps references (integers) to their actual values. The *store* is used to store objects and arrays, which are indexed by a reference that can be used within the *env*. This layer of indirection makes it easier to

```
data Context = Context
  { env :: Env,
    store :: Store,
    out :: [String],
    given :: Val,
    failed :: Bool,
    res :: Val,
    seed :: Ref
  }
```

Listing 8: The *Context* type that stores the configuration data needed by the Mini-Java programming language.

model local variables using the environment, and keeping the environment read-only. This can be linked back to modular structural operational semantics, in which the *env* acts as a contextual entity and the *store* as a mutable entity [27]. The output of the program (that which has been printed) is stored in *out*.

When an expression is executed, the values for the identifiers will be looked up in *env*. This can lead to references, which need to be dereferenced by finding their actual values in the *store*. The *store* can also be modified as part of the execution, for instance when updating the value for a variable. When creating a new object (e.g. `x = new Tree();`), a new reference is created through the use of *seed*, this reference is then used to create an object instance value in *store*, and *env* is updated to link the variable to that reference.

To ensure compatibility with all versions of the explorer framework, the *Context* is designed to not make direct use of Haskell functions to represent methods. Instead, they are represented as an AST within the *Val* data structure. Next to the AST, the *env* at the time of creation of that method is stored to ensure that non-local variables are correctly resolved.

Scheme The second object language integrated within the exploratory programming framework is a simplified dialect of Scheme [28], a functional Lisp-like programming language. This implementation of it supports core features, such as integers, booleans, strings, and lists. Most operations are centred around lists, such as the built-in arithmetic operators (e.g. `(+ 1 2)`) or defining a value (e.g. `(define x 10)`). Booleans can be compared using $<$, and the system output is handled by calling the *display* function. Branching is supported by using the *if* function, which receives a condition, a *then* expression, and a *else* expression. Notably, all functionality within the language is represented by an *Expr* data type. A program consists of a list of these expressions.

The state of a Scheme program, and therefore also the configuration managed by the explorer framework, is represented through the *Context* data structure, which can be found in Listing 9.

```
data Context = Context
  { env :: Env,
    out :: [String]
  }
```

Listing 9: The *Context* type that stores the configuration data needed by the Scheme programming language.

Similarly to Mini-Java, *out* stores the output produced by the program. The *env* here maps the identifiers to their corresponding values. However, unlike Mini-Java, *store* is not used and *env* is updated whenever a change is made to the environment. A value in Scheme can represent an atomic value, like an integer, boolean, string, a list, or a function. Functions are represented as data through an explicit AST, modelled by the *Func* data type. This data type contains built-in primitive operations as well as user-defined lambda abstractions. Crucially, the environment at the time of the creation of a function is not captured, meaning that functions get executed with the current environment (which might have changed). This design choice was made for ease of implementation, and means that where regular Scheme implements lexical scoping, this version of Scheme implements dynamic scoping.

Executing a *Expr* involves resolving symbols in the current *env* and substituting them into the given expression. The expression can then be evaluated. Operations such as *define* directly modify the current

environment, by adding or updating values in *env*.

5.2 Test cases

To be able to test the explorer, test cases must be designed for both the Mini-Java and Scheme languages. The primary focus is on how each version of the explorer behaves in terms of memory usage as they grow in size. However, the run-time performance of the different versions should also be tested to be able to put the memory usage results into perspective. For example, using 2 times less memory, but slowing down the run time by a factor 1000 might not be a good trade-off.

The rest of this section presents the test cases that have been chosen for the experiments.

5.2.1 Memory usage tests

For the memory usage tests, two experiments have been performed. Both of these experiments will use randomly generated expressions to be able to create arbitrary explorer instances that consist of a specific number of nodes, N. By gradually increasing the amount of nodes that the explorer has, the growth of the memory usage can be observed over time. This aims to simulate how the explorer framework would be used in practice. Ideally, expressions taken from actual programs would be used, in order for the experiments to accurately represent practical workloads. However, finding enough suitable programs to perform all experiments would take a lot of time. The approach of using random expressions is deemed an acceptable substitution, sacrificing some accuracy for an easier testing procedure. Important to note here is that for this test case, all versions of the explorer framework use the same randomly generated expressions. This means that the only difference between the versions of the explorer is how they store the data produced by the programs, but they all store the same data.

If the test consisted of simply running N nodes in sequence, the history of the explorer would look like a straight line rather than a tree. While this might not be a problem for the original framework, it could be the case that the shape of the tree influences the performance for some versions. To test if this is the case, the experiments have a built-in parameter p ($0 \leq p \leq 1$). This parameter represents the probability that a jump is made before executing a program with the explorer. In the case of $p = 0.5$, there is a 50% chance that a jump is made. Each of the two experiments will be repeated to vary the value of p. Every value between 0 and 1 is tested with a 0.1 step interval, resulting in each version being tested 11 times.

For the first experiment, each version of the explorer is tested for every value of N from 1 to 100. For the second experiment, a larger step size is taken, testing sizes 100, 200, until the calculation of the memory usage can no longer be performed in a reasonable time (20 minutes).

5.2.2 Run-time performance tests

For the run-time performance, several core operations of the framework will be benchmarked. These benchmarks will only be performed on the versions of the framework that show improved memory usage over the original framework. This is done since the focus is on reducing the memory usage, so only these versions are relevant to be tested more thoroughly.

For the first benchmark, it is tested how long it takes to initialise the explorer and run 100 randomly generated programs in sequence. To be able to better place this benchmark into perspective, it is also performed without using any version of the explorer. This measures the time it takes for the interpreter to run these 100 programs.

For the second benchmark, it is tested how long it takes to execute a single randomly generated program. This benchmark is performed on explorers that have already been initialised and filled with 250 nodes, to avoid taking initialisation time into account.

The third benchmark that is performed is how long it takes to make a random jump. This benchmark is also performed on explorers that are initialised and filled with 250 nodes.

The final benchmark tests the versions of the explorer framework on a single program. This program aims to mimic an actual program written by a user. The program is written in Mini-Java and defines a binary tree data structure along with some operations to manipulate it. The program initialises a binary tree, inserts 500 semi-random integers into the tree, and then checks if all those integers can be found in the tree. While this program consists of only three nodes that need to be added to the explorer, the computational time for the whole program is large. The idea behind this test case is to see how any overhead introduced by the explorer frameworks relates to a computationally heavy program

that represents real-world scenarios. This testcase is harder to replicate in a functional language such as Scheme, so for the sake of brevity and ease of implementation, this test is only implemented for the Mini-Java language.

The revert operation is not tested in these benchmarks, as it is also not tested in the memory usage tests.

5.3 Metrics and measurements

Next, we will discuss how the proposed test cases will be measured.

5.3.1 Memory usage

Choosing an appropriate tool to measure the memory usage of the framework and all other implementations is crucial. However, a fine-grained approach is needed to exactly pinpoint how much memory specific parts of the explorer use. The entire memory usage of the application whilst running could be measured, but this does not give a detailed view. To solve this, the Haskell library *ghc-datasize* will be used. This library provides a feature to measure the memory usage of a specific object in Haskell. This is done by fully evaluating the object, calling the garbage collection, and then performing a heap walk to calculate the exact size in bytes. By applying this to relevant parts of the explorer data structure, the exact size in memory can be calculated.

For the original framework, the *cmap* and *execEnv* are seen as the relevant parts to measure, as they are the parts of the explorer that grow over time. For the other versions, a similar approach is taken, by measuring any new data structures that have been introduced to replace or modify the *cmap* and *execEnv* structures. For version 7, a different approach is taken, as the bulk of the memory usage is not located in memory. Instead, the filesize of the database file used for the explorer will be taken as the memory usage.

One caveat for the size measurement of version 7, is that there is also an in-memory LRU cache that should be measured to get the complete memory usage. However, in order to be able to apply the function that calculates the size in bytes, the structure to be measured must be in the *NFData* typeclass. Although instances for this typeclass can automatically be derived, the LRU cache is imported from a library that does not provide this. Therefore, the LRU cache of version 7 cannot be measured and will not be taken into consideration. This limitation is justified by the fact that the LRU cache is present to remedy the run-time performance and is not essential to the functionality of the framework.

Another limitation in testing the memory usage in this manner is the time it takes per measurement. For small objects, the size can be measured relatively quickly. However, the time the function takes grows rapidly the larger the objects, until it is no longer feasible to wait for the results. This means that the experiments for the memory usage are inherently bound by the resulting size of the objects to be measured. This means that for the second memory usage experiments, the maximum amount of nodes that can be tested will likely differ for each of the object languages.

To ensure the validity of the experiments, each measurement is repeated five times. This way, an average can be taken for each measurement.

5.3.2 Run-time performance

Measuring the run-time performance of the framework accurately is more difficult than the memory usage, as noise can easily influence the results. If the program is interrupted, or the machine on which the experiment is running is busy, the measurements for the run-time will not be accurate.

To ensure the results of the experiments are accurate, the *Criterion* [1] library is used. This library runs a specified benchmark multiple times, measuring the execution time each time. Then, a model is fitted to the gathered datapoints, and the mean of this model is reported alongside the goodness-of-fit. In addition, the mean execution time and standard deviation of the benchmark are given. These two values will be taken as the measurement for the run-time performance experiments performed in this thesis. It is important to note that the exact amount of measurements performed by this framework for each benchmark is not fixed. Instead, a maximum time of 30 seconds is given to run as many measurements as possible for each benchmark.

[1] https://hackage.haskell.org/package/criterion

5.4 Test environment

The experiments will run on a machine with the following specifications:

- CPU: AMD Ryzen 7 7800X3D @ 4.2GHz
- RAM: 32GB DDR5
- OS: Arch Linux on Windows 10 (WSL2)
- Haskell version: GHC 8.8.4

For all experiments, the interpreters are built using the *cabal build* command, without any extra flags. This defaults to the interpreters being built with the *-O1* optimiser flag.

When running the experiments for the run-time performance, noise from other tasks in the environment has been minimised. This means that most programs running on the machine were closed, and only programs essential to the operating system were still running. This should eliminate most noise, but some might still be present in the results for the experiments.

Chapter 6

Results

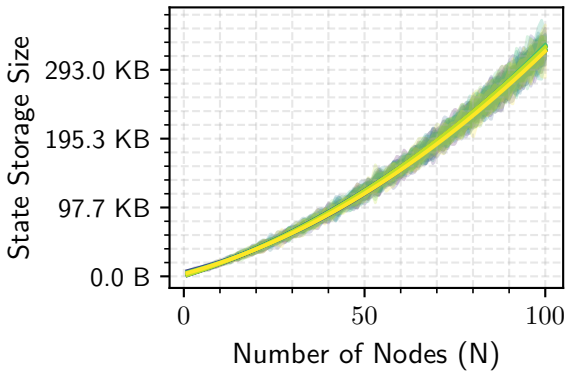
In this chapter, we present the results of our experiments. As all experiments have been performed on two interpreters, we present the results per interpreter.

6.1 Mini-Java

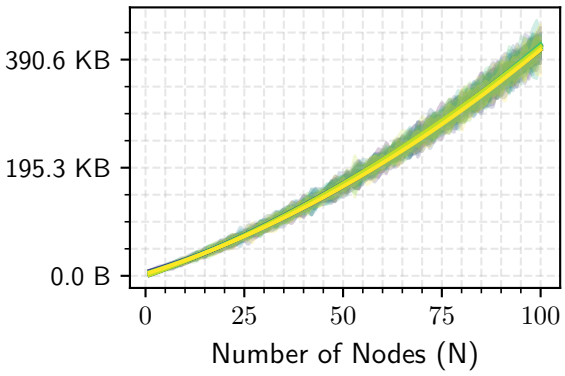
For the Mini-Java language, we first present the memory usage experiments and then the execution time experiments. For the memory usage experiments, we start by showing each version of the framework individually before combining them for comparison.

6.1.1 Individual versions

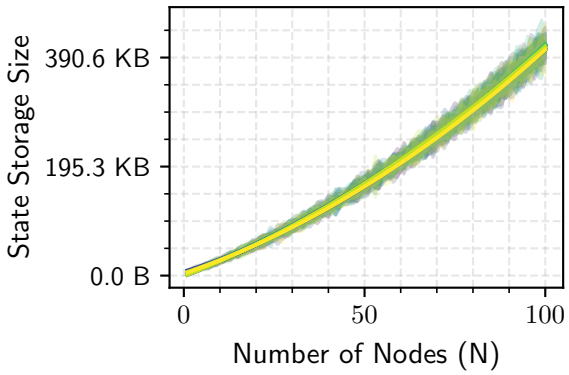
For the first experiment, we have the memory usage per version of the framework over various sizes N. In Figure 6.1, the results of this experiment are presented. For versions 1 to 6, it can be seen that the memory usage of the explorer increases relatively linearly, regardless of the jump probability P. For version 7, it can be seen that the growth pattern of the Explorer varies depending on the jump probability. More specifically, it can be observed that the memory usage for a large number of nodes decreases as the probability of P increases.



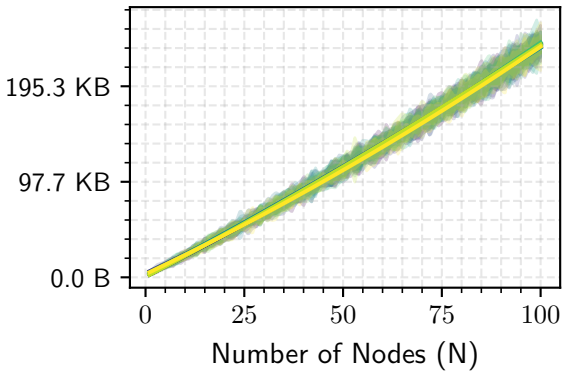
(a) Version 1



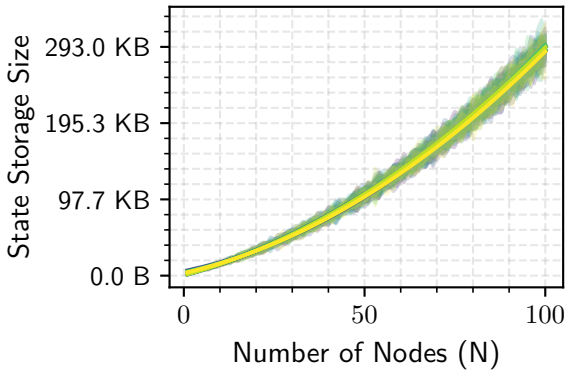
(b) Version 2



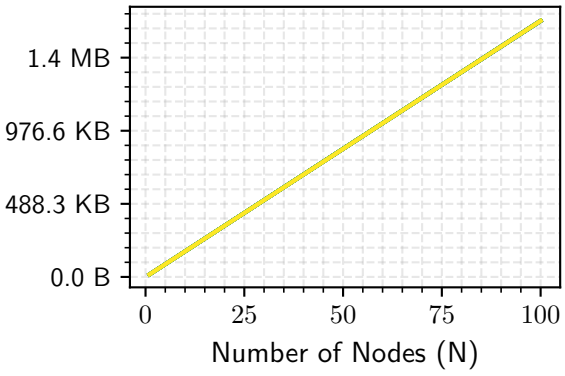
(c) **Version 3**



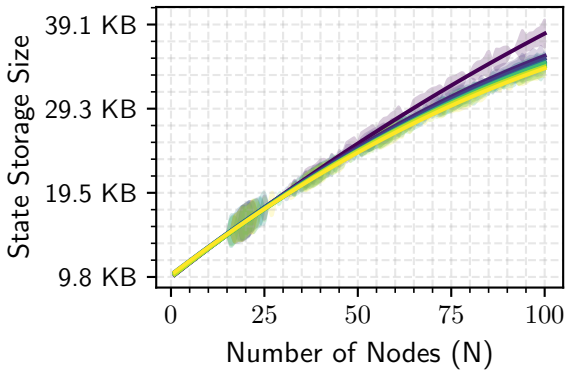
(d) **Version 4**



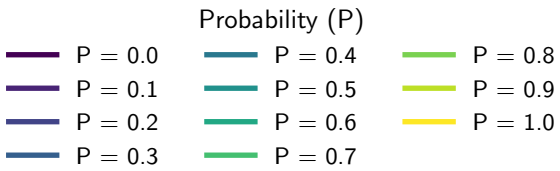
(e) **Version 5**



(f) **Version 6**



(g) **Version 7**



Shared Legend

Figure 6.1: Memory usage over explorer size, for framework versions 1-7.

6.1.2 Versions combined

Next, we show the results of experiment 2, where all versions are combined and tested over a greater range of N. Here, it is important to note that despite having a higher maximum amount of nodes tested than experiment 1, fewer measurements have been performed (see Section 5.2). To avoid having too many lines in one plot, only the results for $P = 0.5$ have been shown. This value has been chosen, as it is deemed likely that any user of the Explorer framework will make some jumps. Moreover, in the previous experiment, it can be seen that for most versions of the framework, $P = 0.5$ is not an outlier. It is important to note that the versions in the legend have been sorted in terms of their memory usage for $N = 300$.

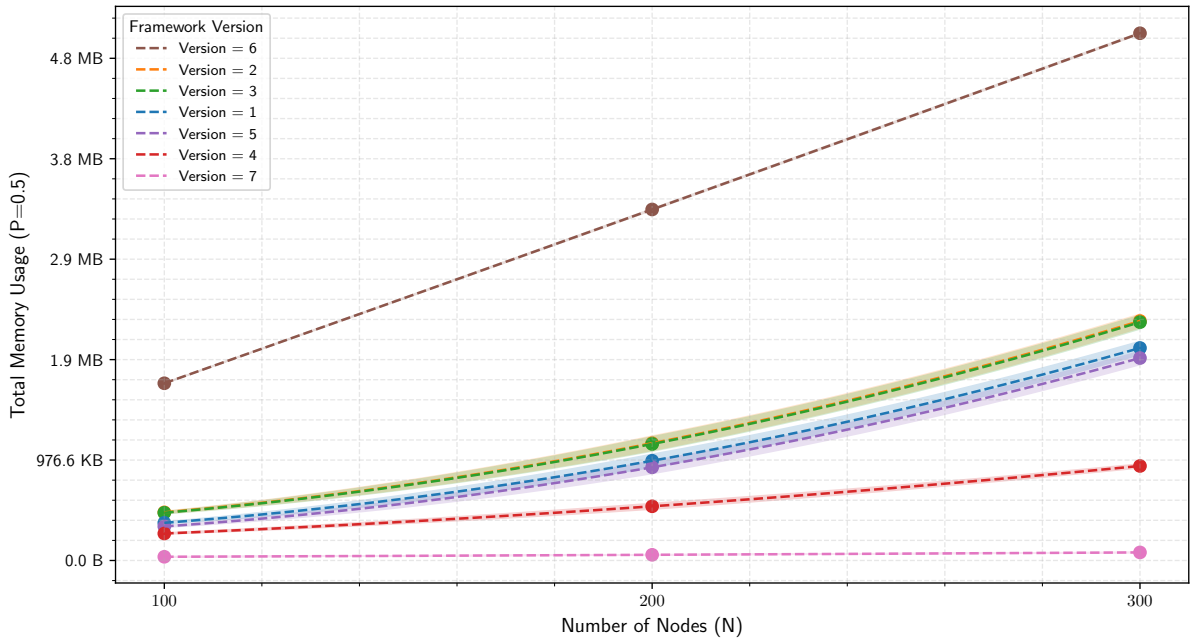


Figure 6.2: Comparison of memory usage for different framework versions.

The results of experiment 2 can be found in Figure 6.2. The first thing to note about this result is that version 6 uses significantly more memory than all other versions. This makes comparing the other versions more difficult. To remedy this, the results of experiment 2 with version 6 omitted can be found in Figure 6.3. Versions 2 and 3 are nearly identical, with version 3 being just a bit smaller. However, both versions have higher memory usage than version 1. Version 5, utilising the minimal tree structure, outperforms version 1 very slightly, with the difference growing over time. Version 4, utilising in-memory patching, performs significantly better than version 1, but is not the best version in this experiment. Version 7 outperforms all other versions, with the memory usage staying near-constant over the size of the explorer. For an explorer of size $N = 300$, version 7 takes up 78.40 kB, whereas version 1 takes up 2.01 MB. This means that in this case, version 7 uses 26 times less memory than version 1.

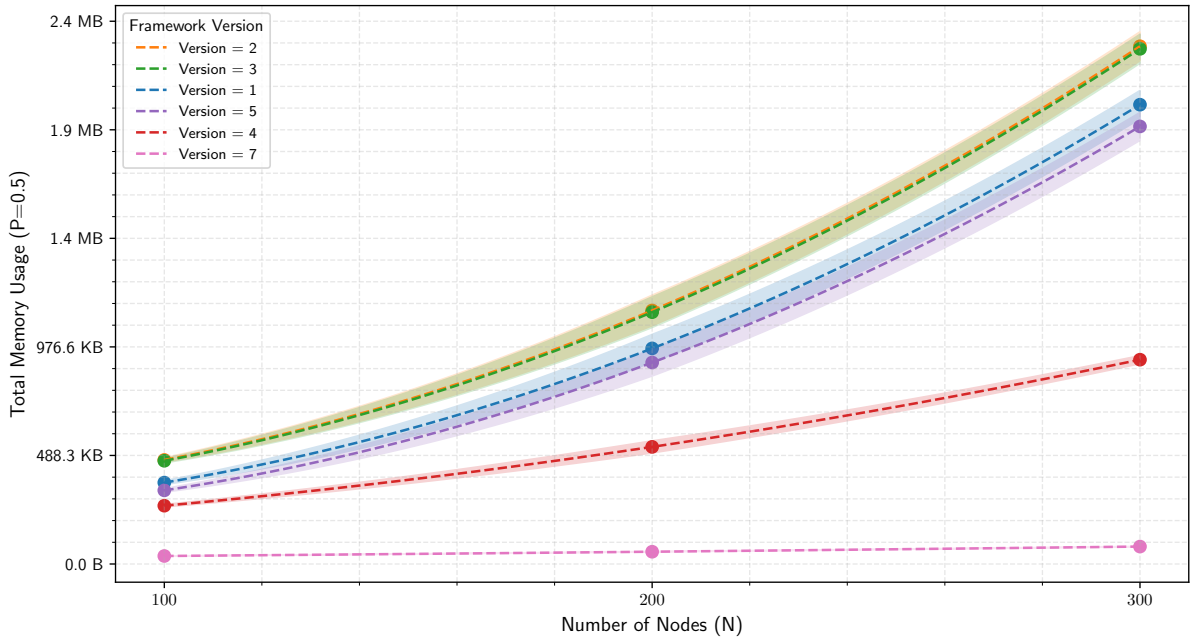


Figure 6.3: Comparison of memory usage for different framework versions.

6.1.3 Run-time performance

Finally, we present the results of experiment 3, where the run-time performance is tested for various versions. The results of this experiment can be found in Table 6.1. Here, *No Explorer* means that the benchmark did not use any version of the explorer framework.

	Benchmark			
Version	Run 100 programs	Random execute	Random jump	Binary tree
No Explorer	$(2.287 \pm 0.065) \times 10^{-3}$	-	-	1.521 ± 0.015
Version 1	$(5.166 \pm 0.142) \times 10^{-3}$	$(1.619 \pm 0.032) \times 10^{-4}$	$(1.428 \pm 0.032) \times 10^{-4}$	1.523 ± 0.007
Version 4	5.039 ± 0.166	$(1.622 \pm 0.040) \times 10^{-4}$	$(1.950 \pm 0.047) \times 10^{-4}$	1.573 ± 0.019
Version 5	$(4.891 \pm 0.149) \times 10^{-3}$	$(1.541 \pm 0.037) \times 10^{-4}$	$(1.420 \pm 0.041) \times 10^{-4}$	1.530 ± 0.008
Version 7	9.304 ± 0.955	$(7.698 \pm 8.199) \times 10^{-2}$	$(3.746 \pm 0.086) \times 10^{-6}$	1.780 ± 0.013

Table 6.1: Run-time performance benchmarks for the Mini–Java language. The values represent the mean ± the standard deviation of the execution time in seconds. Random operations are tested on an explorer of size 250.

Running 100 programs sequentially

For running 100 programs sequentially, it can be seen that versions 1 and 5 perform nearly identically, both being a bit slower than not using any explorer framework. Meanwhile, version 4 is nearly a factor 1000 slower than version 1. Version 7 is a factor of two slower again, almost a factor 2000 slower than version 1.

Randomly executing

For randomly executing one program on an explorer of size 250, it can again be seen that versions 1 and 5 perform nearly identically, but version 4 now also shows similar behaviour. Version 7 remains the slowest, being almost a factor 500 slower than version 1.

Randomly jumping

For executing a random jump on an explorer of size 250, versions 1 and 5 are still nearly identical. Additionally, version 4 shows similar but slightly slower performance. Version 7, however, is now significantly faster than all other versions, being almost a factor 40 faster than version 1.

Running binary tree program

When executing the binary tree program, the performance is very similar for all versions. Versions 1 and 5 are nearly identical, both being approximately 5 milliseconds slower than computing the program without any explorer framework. Meanwhile, version 4 is approximately 50 milliseconds slower. Version 7 is again the slowest program, being approximately 250 milliseconds slower.

This benchmark shows that when the computational time for each execution is large, the relative overhead of the explorer framework versions becomes much smaller. For example, version 4 is now only 3.4% slower, rather than a factor 1000. Similarly, version 7 is 17% slower, rather than the factor 2000 seen before.

6.2 Scheme

Next, we repeat the experiments for the Scheme language.

6.2.1 Individual versions

For this experiment, the results follow the same trend as for the Mini-Java language. As such, the results of this experiment are not shown here for the sake of brevity. The figure can be found in the appendix; see Figure A.1.

6.2.2 Versions combined

Next, the results of the second experiment can be found in Figure 6.4. Here, it can once again be seen that version 6 of the Explorer has significantly higher memory usage, making it difficult to compare the other versions. The results without version 6 can be found in Figure 6.5. Here, the results differ from those for the Mini-Java language. Version 2 and 3 still perform worse than version 1, but version 4 now also has a higher memory usage. On the other hand, version 5 performs relatively better for the Scheme language, having lower memory usage than version 1. Version 7 is still the version with the least amount of memory usage, significantly outperforming all other versions.

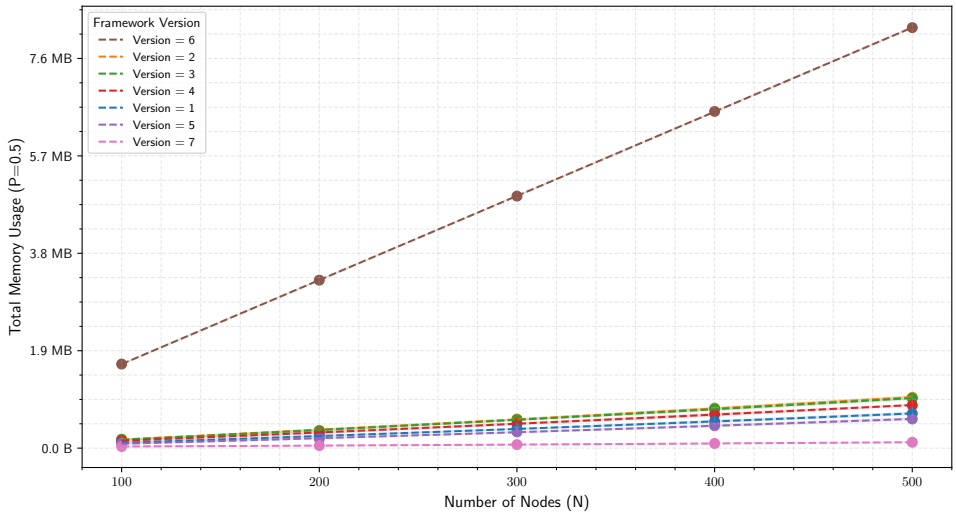


Figure 6.4: Comparison of memory usage for different framework versions.

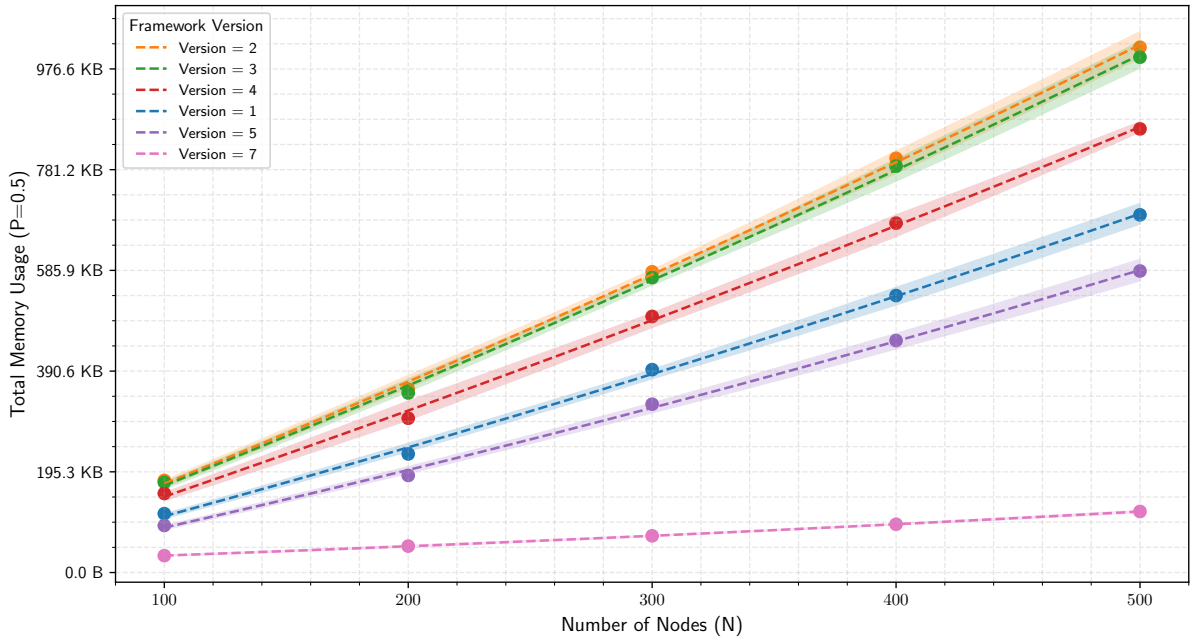


Figure 6.5: Comparison of memory usage for different framework versions.

6.2.3 Run-time performance

Finally, the results of experiment 3 can be found in Table 6.2. Generally, the same findings as for Mini-Java hold. As such, only the differences will be presented here.

	Benchmark		
Version	Run 100 programs	Random execute	Random jump
No Explorer	$(3.875 \pm 0.113) \times 10^{-4}$	-	-
Version 1	$(1.852 \pm 0.033) \times 10^{-3}$	$(1.385 \pm 0.043) \times 10^{-4}$	$(1.060 \pm 0.018) \times 10^{-4}$
Version 4	$(5.867 \pm 0.542) \times 10^{-2}$	$(7.543 \pm 0.018) \times 10^{-5}$	$(1.403 \pm 0.070) \times 10^{-4}$
Version 5	$(1.731 \pm 0.053) \times 10^{-3}$	$(1.277 \pm 0.038) \times 10^{-4}$	$(1.208 \pm 0.029) \times 10^{-4}$
Version 7	3.200 ± 0.021	$(1.859 \pm 0.865) \times 10^{-2}$	$(3.633 \pm 0.006) \times 10^{-6}$

Table 6.2: Run-time performance benchmarks for the Scheme language. The values represent the mean ± the standard deviation of the execution time in seconds. Random operations are tested on an explorer of size 250.

Running 100 programs sequentially

For running 100 programs sequentially, versions 1 and 5 perform relatively worse, as they are both a factor of 5 slower than not using any explorer framework. Furthermore, version 4 is now significantly faster, only a factor of 30 slower than version 1 rather than the previously seen 1000.

Randomly executing

For randomly executing one program on an explorer of size 250, the only difference from Mini-Java is that version 4 is now the fastest, performing almost twice as fast as version 1.

Randomly jumping

For performing random jumps, all findings are the same.

Chapter 7

Discussion

In this chapter, we discuss and interpret the results of the experiments as presented in Chapter 6. We will analyse the performance of each implemented version in comparison to the original framework. This will be done by first focussing on memory usage and then on run-time performance. Furthermore, using the insights gained from this analysis, the subquestions of the main research question will be answered. Finally, the limitations and potential threats to the validity of our findings will be discussed.

7.1 Memory usage results

First, we discuss the results for the memory usage of the framework in the performed experiments. For clarity, the findings for each version are presented separately. Since version 1 is used as a baseline, it will not explicitly be discussed.

7.1.1 Version 2

Finding 7.1: Using a regular tree structure does not perform better than using the graph structure.

The main finding for version 2 can be found in Finding 7.1. As could be seen in the experiments, replacing the *execEnv* with a *parent* and *children* structure did not improve the memory usage of the framework. This is an unexpected result, as it should in theory store less information than the graph representation. However, the performance decrease can be explained by the way this version is implemented. The intmaps in this implementation store a full version of each parent and children nodes. However, nodes in the middle of the tree are both a parent node and a child node. This causes this node to be fully stored in two separate intmaps, which leads to overhead. In hindsight, it might have been better to create an intmap for nodes similar to *cmap*, and use references inside the parent / child relation to refer to nodes.

7.1.2 Version 3

Finding 7.2: A vector representation for the configurations has little to no benefit over an intmap representation.

The main finding for version 3 can be found in Finding 7.2. As could be seen in the experiments, this version exhibits nearly identical memory usage to version 2, being just a little bit lower. This indicates that using a vector to store the configurations can in the best case lead to a slight improvement in memory usage. However, the performed experiments do not perform any revert operations. The performance of this version is likely to degrade compared to version 1 when revert operations are introduced because the vector cannot shrink in size. Although it could be possible to shrink the vector by shifting the data to fill any gaps, the overhead and implementation effort are most likely not worth it for the memory that is saved.

7.1.3 Version 4

Finding 7.3: Storing deltas of configurations in memory can outperform storing them in full.

The main finding for version 4 can be found in Finding 7.3. Version 4 is the only version in which the object language used significantly alters the results of the experiments. For the Mini-Java language, version 4 showed a significant improvement over version 1 (see Figure 6.3). However, for the Scheme language, the same experiment shows that version 4 performs worse than version 1 (see Figure 6.5). This is an unexpected result, as the expectation was that version 4 would show improvements over version 1 regardless of the object language.

A possible explanation for this behaviour is the way the configurations themselves are designed for each language. If one configuration design allows for more granular changes in the JSON representation than the other, the patches created for that design will likely also be smaller. Looking at the internal design of the configurations for the Mini-Java and Scheme languages (see Section 5.1), this seems to be the case. The Mini-Java utilises more mappings than Scheme, which in the JSON structure also gets represented as a map. Changing fields in a nested map in JSON can be done efficiently in the patch, as a path can be followed to find the exact field to change. When there is a change in another kind of structure, it is likely that this entire structure will have to be replaced in the patch. This would mean that the patches for the Mini-Java are more efficient, meaning less memory will be used.

This hypothesis can be tested by checking how large the patches are for each language compared to the full configuration. When this is added to the experiments, it shows that a patch is on average 37.7% the size of a full configuration for the Scheme language, but 28.8% for Mini-Java. This is a significant difference, but not big enough to cause the results of the experiment. Another contributing factor must therefore be the amount of sharing Haskell can apply to the stored patches. Smaller and more efficient patches will likely have a structure that is capable of being stored more efficiently in memory.

The results show that version 4 can be a very good choice for an optimised version of the explorer that still keeps all relevant data in memory. However, this highly depends on how the object language is designed and implemented within Haskell. If the design of the configurations does not result in JSON data that allows for specific changes and therefore small patches, version 4 might end up performing worse than version 1. It is also interesting to note that this behaviour is in part due to the choice of using JSON to generate the patches. It might well be the case that, when using another method (see Section 3.3), the design of the configurations influences the patch size in a totally different way. This shows that while JSON is a good choice for its ease of use, the performance may not be optimal for all users.

7.1.4 Version 5

Finding 7.4: The minimal tree structure outperforms the original graph structure.

The main finding for version 5 can be found in Finding 7.4. Across the experiments for both languages, it can be seen that version 5 outperforms version 1 in terms of memory usage. This is in line with expectations, as this version is designed to store the minimum amount of data required.

7.1.5 Version 6

Finding 7.5: Compressing the configurations in memory is very inefficient.

The main finding for version 6 can be found in Finding 7.5. For this version, it soon became clear that its performance is the worst of all tested versions. Although this is unexpected, the result can be explained. While the compressed configurations are smaller than the configurations themselves, storing them in memory is less efficient. This is likely because Haskell cannot meaningfully apply sharing between different compressed configurations, as their structure has been altered by converting it to a bytestring.

7.1.6 Version 7

Finding 7.6: Optimised on-disk storage of the data offers a significant reduction in memory usage.

The main finding for version 7 can be found in Finding 7.6. This version performed the best across all memory usage experiments, which was not entirely expected. The memory performance was expected to improve over version 1, but the significant difference in performance was not expected.

It is likely that the combination of all optimisation techniques that have been tried in previous versions contributes to this success. By only storing the minimum amount of data needed and compressing where possible, entries in the database are quite small. Moreover, the database is optimised to store data efficiently, so the memory profile of the database file remains small.

An interesting observation in the memory usage of this version is how it varies with the probability of jump P. The higher P, the better the memory usage becomes. This can be explained as being a side effect of the simplistic checkpoint strategy in the implementation. By default, a keyframe node is stored in all 5 nodes, which means the maximum amount of patches that must be applied to reconstruct any configuration is also 5. However, the more jumps performed, the more likely it is that the exploration tree resembles a complete tree. This means that the depth of the tree will be limited, which in turn means that the average distance between any node and a keyframe is likely lower than 5. This leads to the patches being smaller, which leads to the behaviour seen in the results.

A more sophisticated checkpoint strategy could potentially improve the performance in the use case where there are few to no jumps. However, it is also likely that users of an exploratory programming framework will use it to explore different versions of their programs. Given this, the current checkpoint strategy is not a weakness. It also means that choosing $P = 0.5$ to compare all versions is a justified option.

One caveat in the results of version 7 is that the size of the LRU cache is not incorporated. This means that the total memory usage for this version will end up being higher depending on the size of the cache. For a cache of size N, memory usage will consist of at most N configurations plus the overhead added by the data structure. However, because of sharing, the actual memory usage for the cache will likely be lower. Furthermore, the addition of the cache is not essential to the functionality of this version but merely to improve the run-time performance. Therefore, the results of this version as they are now can be used, although one should keep in mind that there is some additional memory usage not shown in them.

7.2 Run-time performance results

Next, the results for the run-time performance experiments are discussed. As before, each tested version is discussed separately, with version 1 being seen as a baseline.

7.2.1 Version 5

Finding 7.7: Version 5 has the same run-time performance for core operations as the original framework, and both come with a slight overhead over not using the framework.

The main finding for version 5 can be found in Finding 7.7. The results for this version are exactly as expected as this version does not change the computational complexity for the core operations. It can be seen that, in all three experiments, version 5 produces very similar results to version 1.

Additionally, these two versions can be compared with the test case of running 100 programs without the use of the exploratory programming framework. Using the framework makes the execution time a factor 2 slower, which is to be expected given the extra overhead that the framework brings. However, it is important to note that the influence of this overhead diminishes when the programs are computationally heavy. This can be seen in the results for the binary tree experiment, as the overhead of versions 1 and 5 becomes 5 milliseconds. This is only 0.33% slower than running the program without any explorer framework.

7.2.2 Version 4

Finding 7.8: Version 4 shows similar run-time performance to the original famework for jumps and random executions, but significantly worse performance for sequential execution.

The main finding for version 4 can be found in Finding 7.8. The results for version 4 are unexpected, especially for the random execution experiment. Creating patches and reconstructing configurations leads to computational overhead, so it would be expected that executing a program would be slower than in the original framework. This overhead can clearly be seen when looking at the execution of a 100 programs, as version 4 is nearly a factor 1000 slower. However, this overhead cannot clearly be seen in the experiment of running a single program on a large explorer, as version 4 is only marginally slower there. This is an unexpected result and cannot be explained properly. More research will be needed to pinpoint what causes this behaviour.

For the random jump experiment, version 4 behaves similarly to version 1. This is also slightly unexpected, as some overhead would be expected here as well. To perform a jump in this version, a reference has to be resolved to a configuration so that the current configuration can be updated. This means that the configuration will need to be reconstructed by applying patches, unless it was stored as a keyframe.

When looking at the binary tree benchmark, the extra overhead introduced by version 4 can be seen. The program takes on average 3.3% longer to run than the original framework. This, again, clearly shows that while the relative overhead per operation might be a lot higher, the absolute overhead when running a few large programs is less significant.

Another interesting observation is the difference in the results for Mini-Java and Scheme. Although the results for the jump experiment are equal, version 4 is the only version in which the execution time results show a significant difference depending on the language. For the Scheme language, version 4 is a factor 10 slower for both sequential and random execution. Considering that the other versions do not have this behaviour, it means that, relatively, version 4 performs better in the context of the Scheme language. This is unexpected, as the memory usage for version 4 was significantly worse in the Scheme language. What exactly causes this behaviour is something that requires further research.

7.2.3 Version 7

Finding 7.9: The overhead of version 7 affects the performance of the run-time for sequential and random executions by a factor of 100 to 2000. However, in the case of computationally heavy programs, this results in only a 17% performance decrease.

The main finding for version 7 can be found in Finding 7.9. For this version, it was already expected that it would have a significantly slower performance than the original framework. This is because of the multiple forms of overhead that are present in this version: creating patches, compressing data, IO interaction with the database, et cetera. The results for the run-time performance experiments confirm this, but are also slightly unexpected. When running 100 programs sequentially, version 7 is around a factor 2000 slower than version 1. However, in the single execution test, it is a factor 500 slower. While it would make sense that initiliasing this version of the framework has additional overhead, it should not have such an impact on the sequential test. A possible explanation for this behaviour is that the single execute test has an explorer that already has executed 250 programs. Both the in-memory cache and the database are warm, which could lead to a single execution taking less time. This would mean that the run-time performance over time improves. A way to test this hypothesis would be to repeat the experiment of sequentially executing N nodes for various sizes and see if the growth in execution time is sublinear.

Fortunately, the results for running the binary tree program are less dramatic. Compared to the original framework, version 7 is 17% slower. This shows that while the overhead per operation is significant, the performance degradation in practice is not as severe.

An interesting outcome for this version is that the standard deviation for the single execution benchmark is higher than its mean for the Mini-Java language. This indicates that the measurements are not normally distributed. While this might seem counterintuitive, this behaviour can be explained. When a program is executed and the corresponding configuration is the same as the previous program, no patch will be made, as the configuration will not be stored in the database. Moreover, if the checkpoint

strategy decides this new configuration is a keyframe, a patch will also not need to be made. In these two scenarios, time is saved by performing less computation. It is likely that this happens frequently enough in the Mini-Java language to cause a spike of measurements with a low execution time. This spike causes the distribution of measurements to shift from a normal distribution to a bimodal one, leading to a mean and standard deviation as seen in the results. Interestingly enough, this behaviour is not directly seen for the Scheme language, which might mean that measurements with low execution time appear less frequently. This could be because of the configuration changing more frequently, which means patches have to be created more often.

Another unexpected finding for version 7 is the speed with which a random jump is made. Being a factor 30 faster than the original framework, while it was thought that it would be slower. As in version 4, the implementation of jumping has additional overhead due to having to reconstruct a node. More research would be needed to investigate whether the result of this experiment is valid and, if so, what could cause this speedup.

7.3 Research questions

Having evaluated the experiments and their results, it is now possible to answer the research questions of this thesis. With the answers to the subquestions, an answer to the main question can be formulated later.

7.3.1 What is the current memory overhead for state storage in the existing exploratory programming framework?

This question can be answered by the measurements performed in the memory usage experiments. The original version of the exploratory programming framework shows a relatively linear growth in overhead, coming from the *cmap* and *execEnv* structures. These structures hold all the data needed by the framework to go back and forth in the history of the explorer. For an explorer of 300 nodes, the memory usage for storing the state is on average around 2 MB for the Mini-Java language, and 700 kB for Scheme. This shows that the overhead for storing the states will vary per language, based on how large the states that need to be stored are.

7.3.2 What methods exist for efficient storage of states, and how do they relate to the exploratory programming framework?

As seen in Chapter 3, numerous methods can be found to ensure efficient storage of states in the Haskell-based exploratory programming framework.

First, the fact that the framework is written in Haskell allows for some inherent optimisations. As discussed in Section 3.1, the immutability of the data structures allows structural sharing to be performed at run-time. This allows parts of the state that are unchanged to be shared between different versions of the state. This provides inherent memory efficiency, as it avoids storing redundant copies of identical sub-components within the state of a program. However, its effectiveness depends on how these states are constructed and managed, and does not handle the case of semantic similarity between distinct objects.

Second, the choice of data structures to represent the execution history of the explorer framework plays a vital role in ensuring efficient memory usage (see Section 3.2). For example, using an *IntMap* to map references to configurations or nodes is generally efficient, especially when the references are sparse. Additionally, the execution history of the framework can be modelled through the use of a minimal tree structure (version 5), as seen in Section 3.2.2). By only storing links to the parents, the amount of overhead per configuration is minimised. This strategy has been shown to have reduced overhead, especially in comparison to a full tree (version 2) or the original graph representation (version 1).

Third, another strategy to reduce storage is to only store differences (deltas) between successive states (see Section 3.3). This method is particularly effective when the changes between states are relatively small compared to their total size. Using *JSON* as an intermediate state to calculate these deltas is a good choice, as it offers a balance between genericity, effectiveness and ease of use.

To manage the computational cost of reconstructing states, periodic storage of full configurations (checkpoints) is essential. The effectiveness of storing deltas is, therefore, tied to the chosen checkpoint strategy. Versions 4 and 7 of the framework utilise this technique and both show improved memory efficiency over the original framework.

Fourth, compression can be used to store the configurations more efficiently, as discussed in Section 3.4. However, its effectiveness varies, as seen in version 6 of the framework, where storing compressed configurations in memory is shown to be inefficient.

Finally, persistent storage is a viable method to reduce the memory usage of the exploratory programming framework (see Section 3.5). It requires storing the data for the explorer on disk and is best used in combination with other methods such as compression or storing only changes in configurations. Version 7 utilises disk storage, along with compression, computing deltas, and an LRU cache to mitigate IO latency. This shows that the aforementioned methods are not mutually exclusive, but instead can be combined to gain further efficiency.

7.3.3 To what extent do assumptions about the object language help to further reduce the memory usage of the framework?

To make some of the optimisations in the framework possible, additional constraints were added for the object languages. This means that an implementation of a language that works with the original framework might not work with all the new versions.

Of the three versions that have been shown to be able to reduce the memory overhead of the explorer, only version 5 is as generic as the original framework. No additional constraints were added, and any object language that works with version 1 will work seamlessly with version 5. However, version 5 is also the version with the least amount of performance increase compared to the original framework.

To make versions 4 and 7 work, the configurations of the object language have to be serialisable to JSON (see Section 3.3). This means that the design of the object language is limited to storing serialisable data in the configurations. This means that implementing functions directly as Haskell functions is not possible. However, this is not to say that functions cannot be implemented, but they have to be embedded or represented through a data structure such as an AST. This is a limitation of the design of these versions of the framework, but it comes with serious performance gains. When assuming that every object language has configurations that can be serialised to JSON, versions 4 and 7 are able to deliver memory overhead that is significantly smaller than the original framework.

7.3.4 How do the implemented state storage methods compare in terms of memory usage and performance trade-offs to the original framework?

As seen earlier in this chapter, the effectiveness of the implemented methods varies greatly. There are a few versions that are able to perform better than the original framework, but there are also versions that make the situation worse. To answer this research question accurately, each version will be compared to the original framework.

Version 2

Version 2 was unable to provide better memory usage than the original framework. This is likely due to data being stored less efficiently than the original graph structure, which leads to more overhead.

Version 3

Version 3 performed nearly identically to version 2, being just a bit smaller for some measurements. This shows that storing the configurations in a vector as opposed to an intmap can save some memory overhead. However, as discussed, this memory saving is not worth the complexity it adds to the framework.

Version 4

Version 4 is hard to compare directly with the original framework, as its memory usage greatly depends on the object language being used. For the Mini-Java language, version 4 offers a great reduction in memory usage. However, for the Scheme language, the memory usage is actually significantly increased.

In terms of run-time performance, version 4 is very interesting. Having a relatively fast single execution time but one of the slowest sequential execution times, it is hard to say whether version 4 is a good choice. Compared to the original framework, running 100 programs can be up to 1000 times slower, but randomly executing a single program takes almost identical time. More research into this behaviour is needed before an accurate comparison can be made with the original framework. However, the binary tree experiment shows that while version 4 slows the execution time, it does so only 3.3%.

Version 5

In terms of memory usage versus performance, version 5 is the most interesting. Due to only changing the way the data is stored, version 5 is able to offer reduced memory usage without influencing the performance of the key operations. This makes it a likely candidate to replace the original framework as it improves the situation with little to no drawbacks. The only drawback of this version is the performance of some of its auxiliary operations, such as converting the data to a tree. However, as discussed previously, these auxiliary functions can be redesigned to improve their performance, making this drawback only minor.

Version 6

Version 6 was the worst performing version of all implementations, showing significantly more memory overhead. This is likely due to Haskell being unable to efficiently store the bytestrings that are produced by compressing the configurations.

Version 7

Version 7 is the final implementation, combining several of the previously implemented methods with on-disk storage. While its memory usage is the best seen across all methods, its run-time performance is the worst. Whether or not this version is a success depends on the priorities of the user, and what kind of programs it will be used for.

If the sole goal is to reduce the memory usage of the original framework, version 7 is the best way to do it. In the experiments that have been performed, version 7 had memory usage that was up to 26 times less than the original framework (see Section 6.1). No other implementation was able to outperform version 7 in any test where the explorer contained more than 10 nodes. When the explorer is very small, the overhead of the database is greater than the savings. This overhead quickly disappears as the explorer grows, as the rate at which the memory usage increases for version 7 is incredibly low.

However, if run-time performance is also crucial, version 7 might not be the best choice. As seen in the experiments, initialising an explorer and running 100 programs can be up to a factor 1000 slower than in the original framework. This can be a dealbreaker if a lot of small programs need to be run, as the overhead for this version quickly adds up. However, if the programs that need to be run are large and computationally heavy, the overhead becomes relatively smaller. As an example, in the binary tree program, version 7 is 17% slower, which is significantly better than the potential factor 1000.

Moreover, further research could investigate and improve the run-time performance of this version. One possibility is to combine the memory performance of this version with the run-time performance of the other versions. This could be achieved by using a feature of SQLite to store a database in memory [1]. If run-time performance can be increased while maintaining memory overhead as is, this version of the framework shows serious potential.

Another aspect of this version to consider is the complexity of the implementation and its impact on the end user. This version has been designed in a way to limit the burden on the end user by making sure that most implementation details are hidden by the framework. By choosing JSON as the technique to patch the configurations, the users only need to derive a generic implementation for their configurations. However, users must design the data they wish to store in a specific way, taking special care that their data can be converted to JSON. Additionally, this version of the framework forces users to make use of the IO monad, as the explorer itself consists of IO due to the database.

To conclude, version 7 is an excellent choice in terms of memory usage, but its run-time performance and the slight complexity added to the end-user's experience can be a deal-breaker.

7.4 Threats to validity

Finally, we address any threats to validity.

7.4.1 Research method

One threat to the validity of this thesis is the research method. A literature study was conducted to identify potential methods to reduce the memory consumption of the exploratory programming framework. However, it could be the case that the methods that were found and implemented were not the

[1] https://www.sqlite.org/inmemorydb.html

best choice. There could be other methods that are more effective for this use case that have not been identified by the literature study.

7.4.2 Representativeness of testcases

Another issue that could threaten the results of this thesis is the testcases that have been used for the experiments. The testcases currently use randomly generated expressions that mimic the interaction that a user could have with the explorer. While it has been ensured that the randomly generated expressions are valid, it cannot be guaranteed that they are actually logical. It could well be that real interaction with the explorer follows other patterns than the ones tested, which in turn could lead to different test results.

To remedy this for the run-time performance measurements, the binary tree testcase has been added. This testcase aims to represent a practical use case of the Mini-Java language, against which the other testcases can be compared.

7.4.3 Correctness of implementations

In several of the new versions of the framework, the data sent to the explorer by the user is stored in some altered way. Whether it is through compression, patching, or storing it on-disk, the data has to be reconstructed to be used again. This comes at the risk of the reconstructed data not being equal to the data the user originally sent. If this is the case, the explorer could crash or lose critical information about the session.

To ensure that the implementations do not have this problem, a correctness check has been performed. This check consists of building two equivalent explorers of size 250, one using the original framework and one using another implementation version. Then, the *toTree* function is called for both versions, converting them to two tree structures. Since these tree structures contain the full configurations, they all have to have been reconstructed by the explorer framework. Then it can be checked if the two returned trees are equal to each other or if some data has been corrupted. This check has been run multiple times for each version of the framework, using both the Mini-Java and Scheme languages, and showed no problems.

Chapter 8

Related work

In terms of related work, a main area of interest is back-in-time or omniscient debugging. These kinds of debugging tools record full execution histories and allow users to go back in time through past program states. Some of the earlier work in this area worked by recording every change in the state of a program while running it [29]. This allows users to step back and forth in the program execution, without having to re-run the program each time. The execution history as seen in the exploratory programming framework as presented by [1] draws inspiration from this concept.

Like in exploratory programming, back-in-time debugging faces challenges with regard to memory usage. Storing all previous states of a program comes with significant memory overhead. A balance between memory efficiency and run-time performance is important, just as seen in this work on exploratory programming. If the debugging tool becomes too slow, it is no longer useful in practice. Similarly, if the memory usage of the debugging tool becomes too large, it also limits the usability. The authors of [30] noted this and looked at reducing the amount of data that is stored while maintaining the core functionality. This can be related to version 5 of the framework in this research, where a minimal tree structure is used to minimise the amount of data stored. Later work further improved memory usage by using on-disk approaches combined with indexing for fast querying [31, 32]. The final version of the exploratory programming framework is loosely based on this idea, as it also utilises an on-disk database combined with indexing.

An area where omniscient debugging can be used is in the logging of data for programs running in production. This is particularly useful in the case of a bug experienced by a user, as it allows developers to pinpoint where something went wrong in the execution of the program. However, these traces can become quite large, which means storing a lot of them can take up massive amounts of memory usage. The authors of [33] looked at limiting the size of a single execution trace, by limiting the amount of times repetitive operations are stored. This approach could potentially work for the exploratory programming framework, for instance, if the same program is run two times in a row. However, this would require detecting if the program that will be executed has monadic side effects, since in that case both programs and their effects need to be stored.

While omniscient debugging considers the execution history to be a straight line, this is not the case for exploratory programming. A more specific area of omniscient debugging is multiverse debugging [34], which more closely resembles the graph-like structures for the execution history. Multiverse debugging is a form of omniscient debugging, specifically designed for non-deterministic programs. This approach allows developers to observe and interact with all possible execution paths that a concurrent program can take.

A problem that occurs in multiverse debugging is the complexity of finding a user-specified breakpoint. In omniscient debugging, the execution history can be traversed until the breakpoint specified by the user is found. In multiverse debugging, there can be numerous possible execution paths that the non-deterministic concurrent program can take. To find a specific breakpoint, in the worst case, the whole state-space (the set of possible states) needs to be searched. The state-space can become very large, or even infinite in some cases.

An approach to reducing the state-space is to define reduction policies, which specify equivalence classes for configurations in the state-space [35]. These reduction policies tell the debugger when two configurations are similar enough that they can be treated as equal, allowing them to collapse into one. This significantly reduces the size of the state-space, and makes multiverse debugging more practical.

This approach of applying equivalence classes to reduce the amount of states that need to be stored

can be related to model checking [36], and is also promising for the exploratory programming framework. Model checking is a technique for verifying programs that consist of finite state to ensure that it behaves correctly. In model checking, the main challenge is that the state space of a program grows very rapidly, also known as the *state explosion problem* [37, 38]. Adding reduction policies through equivalence classes to the exploratory programming framework could be a very interesting approach to lowering the memory usage and is left for future potential research.

Lastly, the work of [39] presents a generic omniscient debugger that can be implemented for domain-specific languages, similarly to how the work of [1] presents a generic back-end for exploratory programming. Developers can integrate their programming language with the generic debugger, in order to easily facilitate omniscient debugging for their language.

Chapter 9

Conclusion

In this thesis, we have explored the methods that can be used to reduce the memory usage of an exploratory programming framework. We have seen that methods like minimising the data that is stored, or in-memory patching can offer moderate improvements in memory consumption with minimal drawbacks in the run-time performance. Storing data compressed on disk can achieve significant improvements in memory consumption, but come at a significant cost to execution time for some operations. The choice of method to apply depends heavily on the programs that are executed and the specific requirements of the user, in terms of how much memory usage they want to save versus the amount of performance they are willing to sacrifice.

9.1 Future work

If given more time, there are numerous topics within this thesis that deserve additional research. In this section, we will go over the most important topics that would be tackled in future work, in order from most to least significant.

9.1.1 Additional experiments

Some results from the experiments were unexpected, such as for version 4. Its memory performance depended heavily on the object language, and its run-time performance seemed inconsistent. Further research could perform more experiments on version 4 to pinpoint what causes this behaviour and gain a better understanding of how exactly the object language influences the performance.

9.1.2 In-memory database

During the final research for this work, it was discovered that SQLite offers a feature to store its database in memory. The Haskell library used within the framework also supports this feature, which means that it is possible to use this feature in the on-disk storage version. This would theoretically keep the benefit of the optimised memory usage that version 7 has, but with improved run-time performance due to keeping everything in memory. It would be interesting to see how this would work and if it can bridge the gap between good memory and good run-time performance.

Acknowledgements

I would like to thank Damian Frolich and Thomas van Binsbergen. Writing this thesis was not an easy process, but their seemingly endless patience and support has helped me through it.

Bibliography

[1] D. Frolich and L. T. van Binsbergen, "A generic back-end for exploratory programming," in *Trends in Functional Programming*, V. Zsók and J. Hughes, Eds., Cham: Springer International Publishing, 2021, pp. 24–43, ISBN: 978-3-030-83978-9. DOI: 10.1007/978-3-030-83978-9_2.

[2] M. B. Kery, A. Horvath, and B. Myers, "Variolite: Supporting exploratory programming by data scientists," in *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, ser. CHI '17, New York, NY, USA: Association for Computing Machinery, 2017, pp. 1265–1276, ISBN: 9781450346559. DOI: 10.1145/3025453.3025626.

[3] J. F. Pimentel, L. Murta, V. Braganholo, and J. Freire, "A large-scale study about quality and reproducibility of jupyter notebooks," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, 2019, pp. 507–517. DOI: 10.1109/MSR.2019.00077.

[4] 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*, ser. CHI '18, Montreal QC, Canada: Association for Computing Machinery, 2018, pp. 1–12, ISBN: 9781450356206. DOI: 10.1145/3173574.3173606.

[5] T. Kluyver *et al.*, "Jupyter notebooks–a publishing format for reproducible computational workflows," in *Positioning and power in academic publishing: Players, agents and agendas*, Amsterdam: IOS press, 2016, pp. 87–90. DOI: 10.3233/978-1-61499-649-1-87.

[6] M. Beth Kery and B. A. Myers, "Exploring exploratory programming," in *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 2017, pp. 25–29. DOI: 10.1109/VLHCC.2017.8103446.

[7] 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*, ser. Onward! 2020, Virtual, USA: Association for Computing Machinery, 2020, pp. 84–100, ISBN: 9781450381789. DOI: 10.1145/3426428.3426917.

[8] C. Okasaki, *Purely functional data structures*. Cambridge University Press, 1998.

[9] Z. M. Ariola and Arvind, "Properties of a first-order functional language with sharing," *Theoretical Computer Science*, vol. 146, no. 1, pp. 69–108, 1995, ISSN: 0304-3975. DOI: 10.1016/0304-3975(94)00185-L.

[10] V. Bhaskaran and K. Konstantinides, *Image and video compression standards: algorithms and architectures* (The Springer International Series in Engineering and Computer Science), 2nd ed. Springer, 1997, vol. 408. DOI: 10.1007/978-1-4615-6199-6.

[11] J. Gibbons and N. Wu, "Folding domain-specific languages: Deep and shallow embeddings (functional pearl)," in *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP '14, Gothenburg, Sweden: Association for Computing Machinery, 2014, pp. 339–347, ISBN: 9781450328739. DOI: 10.1145/2628136.2628138.

[12] S. Erdweg, T. Szabó, and A. Pacak, "Concise, type-safe, and efficient structural diffing," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '21)*, Virtual, Canada: ACM, 2021, pp. 406–419. DOI: 10.1145/3453483.3454052.

[13] E. Lempsink, "Generic type-safe diff and patch for families of datatypes," Master's thesis, Universiteit Utrecht, 2009. [Online]. Available: http://eelco.lempsink.nl/thesis.pdf (visited on 06/07/2025).

[14] C. Percival. "bsdiff—binary diff/patch utility." (2003), [Online]. Available: https://www.daemonology.net/bsdiff/ (visited on 05/12/2025).

[15] P. C. Bryan and M. Nottingham, *JavaScript Object Notation (JSON) Patch*, RFC 6902, Apr. 2013. DOI: 10.17487/RFC6902.

[16] J.-l. Gailly and M. Adler. "Zlib compression library." (2004), [Online]. Available: http://www.dspace.cam.ac.uk/handle/1810/3486 (visited on 05/13/2025).

[17] Y. Collet and M. Kucherawy, *Zstandard compression and the 'application/zstd' media type*, RFC 8878, Feb. 2021. DOI: 10.17487/RFC8878.

[18] A. P. Maulidina, R. A. Wijaya, K. Mazel, and M. S. Astriani, "Comparative study of data compression algorithms: Zstandard, zlib & lz4," in *International Conference on Science, Engineering Management and Information Technology*, Springer, 2023, pp. 394–406. DOI: 10.1007/978-3-031-72284-4_24.

[19] SQLite Consortium. "Sqlite home page." Last modified 2025-05-06. (2025), [Online]. Available: https://www.sqlite.org/ (visited on 05/13/2025).

[20] M. Erwig, "Inductive graphs and functional graph algorithms," *Journal of Functional Programming*, vol. 11, no. 5, pp. 467–492, 2001. DOI: 10.1017/S0956796801004075.

[21] W. Szpankowski, "Patricia tries again revisited," *J. ACM*, vol. 37, no. 4, pp. 691–711, Oct. 1990, ISSN: 0004-5411. DOI: 10.1145/96559.214080.

[22] L. T. van Binsbergen *et al.*, "A language-parametric approach to exploratory programming environments," in *Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering*, ser. SLE 2022, Auckland, New Zealand: Association for Computing Machinery, 2022, pp. 175–188, ISBN: 9781450399197. DOI: 10.1145/3567512.3567527.

[23] Y. Collet, *zstd 1.5.1 Manual*, 1.5.1, Facebook, Inc., 2025. [Online]. Available: https://facebook.github.io/zstd/zstd_manual.html (visited on 05/13/2025).

[24] A. W. Appel and J. Palsberg, *Modern Compiler Implementation in Java, 2nd Edition*. Cambridge, UK; New York, NY, USA: Cambridge University Press, 2002, p. 501, ISBN: 0-521-82060-X.

[25] A. W. Appel and J. Palsberg. "The minijava project," Cambridge University Press. (2002), [Online]. Available: https://www.cambridge.org/us/features/052182060X/ (visited on 06/09/2025).

[26] J. Forden, A. Gebhard, M. Berner, and D. Brylow, "Minijava on risc-v: A game of global compilers domination," in *Proceedings of the Workshop Dedicated to Jens Palsberg on the Occasion of His 60th Birthday*, ser. JENSFEST '24, Pasadena, CA, USA: Association for Computing Machinery, 2024, pp. 21–29, ISBN: 9798400712579. DOI: 10.1145/3694848.3694854.

[27] P. D. Mosses, "Modular structural operational semantics," *The Journal of Logic and Algebraic Programming*, vol. 60-61, pp. 195–228, 2004, Structural Operational Semantics, ISSN: 1567-8326. DOI: https://doi.org/10.1016/j.jlap.2004.03.008. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S156783260400027X.

[28] R. K. Dybvig, *The Scheme programming language*. Cambridge, MA: MIT Press, 2009, ISBN: 978-0-262-51298-5.

[29] B. Lewis, *Debugging backwards in time*, 2003. arXiv: cs/0310016 [cs.SE].

[30] A. Lienhard, T. Gîrba, and O. Nierstrasz, "Practical object-oriented back-in-time debugging," in *ECOOP 2008 – Object-Oriented Programming*, J. Vitek, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 592–615, ISBN: 978-3-540-70592-5. DOI: 10.1007/978-3-540-70592-5_25.

[31] G. Pothier, É. Tanter, and J. Piquer, "Scalable omniscient debugging," *SIGPLAN Not.*, vol. 42, no. 10, pp. 535–552, Oct. 2007, ISSN: 0362-1340. DOI: 10.1145/1297105.1297067.

[32] G. Pothier and É. Tanter, "Summarized trace indexing and querying for scalable back-in-time debugging," in *European Conference on Object-Oriented Programming*, Springer, 2011, pp. 558–582. DOI: 10.1007/978-3-642-22655-7_26.

[33] K. Shimari, T. Ishio, T. Kanda, and K. Inoue, "Near-omniscient debugging for java using size-limited execution trace," in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2019, pp. 398–401. DOI: 10.1109/ICSME.2019.00068.

[34] C. Torres Lopez, R. Gurdeep Singh, S. Marr, E. Gonzalez Boix, and C. Scholliers, "Multiverse Debugging: Non-Deterministic Debugging for Non-Deterministic Programs," in *33rd European Conference on Object-Oriented Programming (ECOOP 2019)*, vol. 134, Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019, 27:1–27:30, ISBN: 978-3-95977-111-5. DOI: 10.4230/LIPIcs.ECOOP.2019.27.

[35] M. Pasquier, C. Teodorov, F. Jouault, M. Brun, L. L. Roux, and L. Lagadec, "Practical multiverse debugging through user-defined reductions: Application to uml models," in *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems*, ser. MODELS '22, Montreal, Quebec, Canada: Association for Computing Machinery, 2022, pp. 87–97, ISBN: 9781450394666. DOI: 10.1145/3550355.3552447.

[36] E. M. Clarke, "Model checking," in *Foundations of Software Technology and Theoretical Computer Science*, S. Ramesh and G. Sivakumar, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 54–56, ISBN: 978-3-540-69659-9.

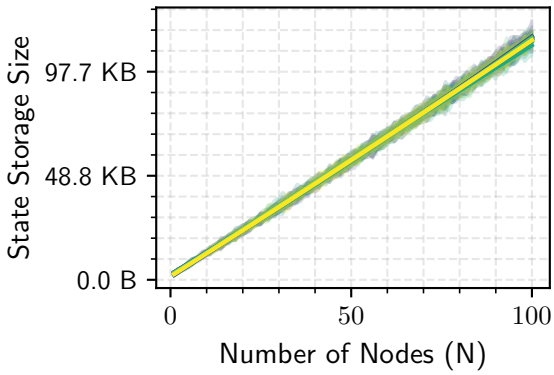
[37] E. M. Clarke, W. Klieber, M. Nováček, and P. Zuliani, "Model checking and the state explosion problem," in *Tools for Practical Software Verification: LASER, International Summer School 2011, Elba Island, Italy, Revised Tutorial Lectures*, B. Meyer and M. Nordio, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 1–30, ISBN: 978-3-642-35746-6. DOI: 10.1007/978-3-642-35746-6_1.

[38] A. Valmari, "The state explosion problem," in *Lectures on Petri Nets I: Basic Models: Advances in Petri Nets*, W. Reisig and G. Rozenberg, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 429–528, ISBN: 978-3-540-49442-3. DOI: 10.1007/3-540-65306-6_21.

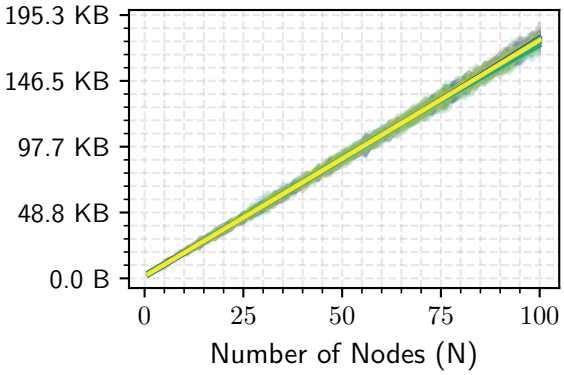
[39] E. Bousse, D. Leroy, B. Combemale, M. Wimmer, and B. Baudry, "Omniscient debugging for executable dsls," *Journal of Systems and Software*, vol. 137, pp. 261–288, 2018. DOI: 10.1016/j.jss.2017.11.025.

Appendix A

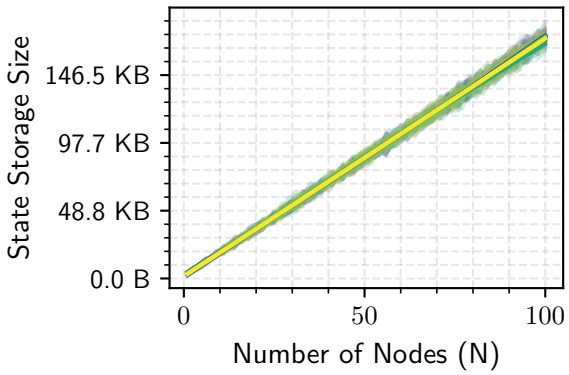
Scheme figures



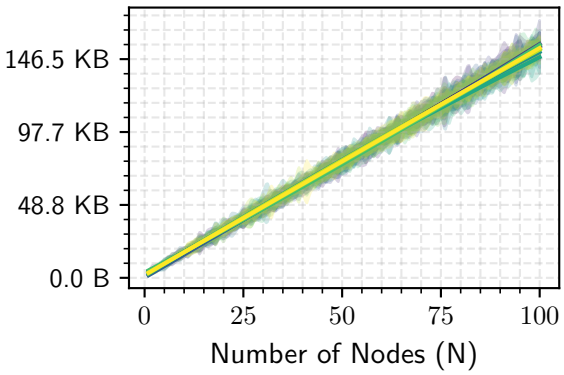
(a) Version 1



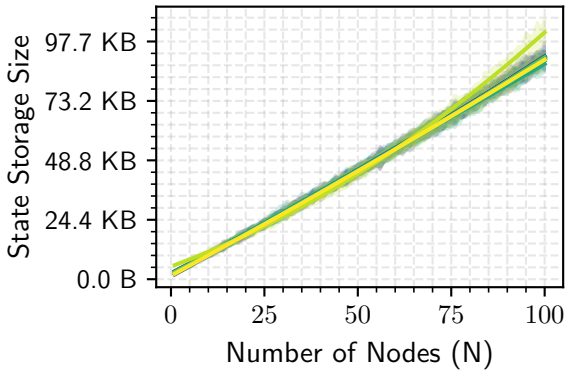
(b) Version 2



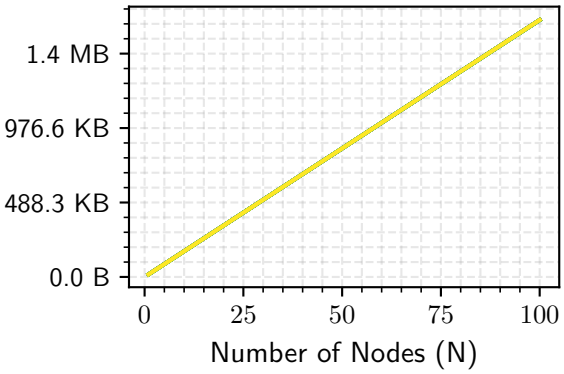
(c) Version 3



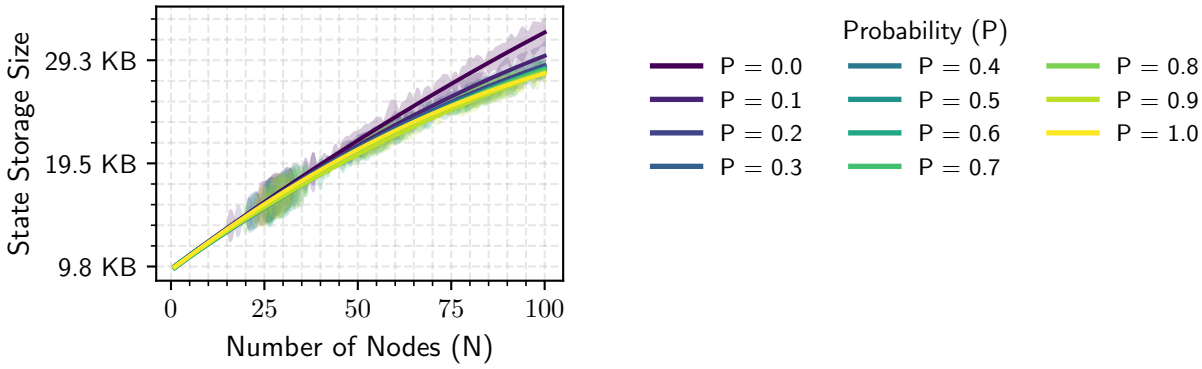
(d) Version 4



(e) Version 5



(f) Version 6



(g) Version 7 **Shared Legend**

Figure A.1: Memory usage over explorer size, for framework versions 1-7.

Appendix B

Source code

The source code for this thesis can be found in the following GitHub repository:
https://github.com/Olaf-Erkemeij/exploring_interpreters