

UNIVERSIDADE FEDERAL DE MINAS GERAIS
Instituto de Ciências Exatas
Programa de Pós-Graduação em Ciência da Computação

Caio Vinícius Raposo Ribeiro

Memoization of Mutable State

Belo Horizonte
2024

Caio Vinícius Raposo Ribeiro

Memoization of Mutable State

Final Version

Thesis presented to the Graduate Program in Computer Science of the Federal University of Minas Gerais in partial fulfillment of the requirements for the degree of Master in Computer Science.

Advisor: Fernando Magno Quintão Pereira

Belo Horizonte
2024

Ribeiro, Caio Vinícius Raposo.

R484m Memoization of mutable state [recurso eletrônico] /
Caio Vinícius Raposo Ribeiro. – 2024.

1 recurso online (60 f. il., color.) : pdf.

Orientador: Fernando Magno Quintão Pereira.

Dissertação (mestrado) - Universidade Federal de Minas
Gerais, Instituto de Ciências Exatas, Departamento de Ciência
da Computação.

Referências: f. 48-46.

1. Computação – Teses. 2. Programação orientada a objetos
(Computação) – Teses. 3. Linguagens de programação – Teses
I. Pereira, Fernando Magno Quintão. II. Universidade Federal
de Minas Gerais, Instituto de Ciências Exatas, Departamento
de Ciência da computação. III. Título.

CDU 519.6*31(043)



UNIVERSIDADE FEDERAL DE MINAS GERAIS

MEMOIZATION OF MUTABLE STATE

CAIO VINÍCIUS RAPOSO RIBEIRO

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

Prof. Fernando Magno Quintão Pereira - Orientador
Departamento de Ciência da Computação - UFMG

Prof. André Rauber Du Bois
Centro de Desenvolvimento Tecnológico - UFPel

Prof. Haniel Moreira Barbosa
Departamento de Ciência da Computação - UFMG

Belo Horizonte, 21 de junho de 2024.



Documento assinado eletronicamente por **Fernando Magno Quintão Pereira, Professor do Magistério Superior**, em 05/08/2024, às 16:00, conforme horário oficial de Brasília, com fundamento no art. 5º do [Decreto nº 10.543, de 13 de novembro de 2020](#).



Documento assinado eletronicamente por **André Rauber Du Bois, Usuário Externo**, em 06/08/2024, às 13:41, conforme horário oficial de Brasília, com fundamento no art. 5º do [Decreto nº 10.543, de 13 de novembro de 2020](#).



Documento assinado eletronicamente por **Haniel Moreira Barbosa, Professor do Magistério Superior**, em 06/08/2024, às 19:25, conforme horário oficial de Brasília, com fundamento no art. 5º do [Decreto nº 10.543, de 13 de novembro de 2020](#).



A autenticidade deste documento pode ser conferida no site https://sei.ufmg.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0, informando o código verificador **3311405** e o código CRC **FB92A685**.

Aos meus pais, Evandro e Elinete.

Acknowledgments

I would like to express my deepest gratitude to my parents, for their unwavering support throughout this journey. I owe a special thanks to my advisor, Fernando Magno Quintão Pereira, for his patience and continuous encouragement. Even during moments of self-doubt, Fernando always pushed me forward and showed great care. He taught me what it means to be a researcher. I also thank all my labmates from The Dragons Nest and to my friends Everaldo Andrade, Lucas Peres, and Michael Canesche, your support was fundamental for me to complete this work.

This research was supported by grant 88887.686766/2022-00 from the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES), and I am very grateful of their support over the past two years.

“Those who cannot remember the past are condemned to repeat it.”
(George Santayana)

Resumo

Memoização é uma otimização que consiste em armazenar em cache os resultados das funções para evitar recomputações de chamadas repetidas. Esta técnica é incorporada nativamente em algumas linguagens de programação e implementada como um padrão de design em outras. No entanto, apesar de sua popularidade, a memoização tem limitações. Em particular, objetos mutáveis não podem ser armazenados em cache, pois se um objeto memoizado sofrer mutação, essa modificação poderá afetar todas as instâncias memoizadas dele. Esta dissertação apresenta uma técnica para resolver esta restrição usando ponteiros de posse compartilhada para distinguir entre objetos memoizados e não memoizados. Se um objeto memoizado for modificado, esse objeto será removido da tabela de memoização e todos os seus aliases serão atualizados por meio do ponteiro compartilhado. Implementamos esta técnica no ambiente de execução da linguagem de programação Hush. Nossa implementação não incorre em penalidades em objetos não memoizados e adiciona sobrecarga mínima aos objetos armazenados em cache. Para demonstrar a correteza dessa abordagem, nós a formalizamos na linguagem de modelagem Alloy. Esta especificação certifica que um objeto memorizado permanece assim, a menos que seja modificado.

Palavras-chave: Memoização; Programação Orientada a Objetos; Cache de Funções.

Abstract

Memoization is an optimization that consists of caching the results of functions to avoid recomputations of repeated calls. This technique is natively built into some programming languages and implemented as a design pattern in others. Yet, in spite of its popularity, memoization has limitations. In particular, mutable objects cannot be cached, because if a memoized object is mutated, then this modification might have an effect on all the memoized instances of it. This dissertation presents a technique to address this constraint by using shared-ownership pointers to distinguish between memoized and non-memoized objects. If a memoized object is modified, then this object is removed from the memoization table, and all its aliases are updated through the shared pointer. We have implemented this technique into the runtime environment of the Hush programming language. Our implementation incurs no penalty on non-memoized objects and adds minimum overhead to cached ones. To demonstrate the correctness of this approach, we have formalized it in the Alloy modeling language. This specification certifies that a memoized object remains so unless it is modified.

Keywords: Memoization; Object-Oriented Programming; Function Caching.

List of Figures

2.1	Execution time of <i>knapsack</i> on different y-axis scales.	20
3.1	Distinction between non-memoized and memoized objects. The latter are accessed through a shared pointer, and stored in the memoization table unless modified.	23
3.2	(a) Creation of memoized object. (b) Memoized aliases. (c) Logical aliases. (d) Copy-on-write semantics.	24
3.3	Rules for creating objects. The environment “env” associates variable names with values.	25
3.4	Rules to access object properties.	25
3.5	Rules to modify object properties.	26
4.1	Instance found by the Alloy Analyzer.	28
5.1	Running time of different memoized programs in functional programming style.	35
5.2	Running time of Max Subarray Sum	36
5.4	Example that illustrates the heavy impact of deep hashing on the memoization of complex objects.	37
5.5	Execution time of Lines using different types of keys.	38
5.6	Execution time of the <i>levenshtein</i> benchmark, with inputs ranging from 0 to 30. Builtin memoization is using the <code>#[unsafe_ignore_trace]</code> macro from the <code>Gc</code> library.	40
5.7	Flamegraph of functions executed in the Hush interpreter for the <i>levenshtein</i> benchmark, with purple indicating functions related to garbage collection. In (a), up to 50% of the total execution time is concentrated in garbage collection routines, while (b) spends only 1.8% of the total time in these functions. . . .	40
5.3	Execution time and hit rate of object-oriented benchmarks.	43

Contents

1	Introduction	12
2	Background	15
2.1	Automatic Memoization	15
2.2	The Challenge of Mutability	16
2.3	The Hush Programming Language	18
2.4	Theorem Provers and Model Checkers	19
3	Shared Ownership of Memoized Values	22
3.1	Shared-Ownership Pointers	22
3.2	Object Creation	24
3.3	Property Accesses	25
3.4	The Copy-On-Write Semantics	25
4	Formal Specification	27
4.1	Introduction to Alloy	27
4.2	The Memoization Model	28
4.2.1	Properties	29
5	Empirical Evaluation	31
5.1	RQ1: Traditional Function Memoization	32
5.2	RQ2: Memoization of Objects	34
5.3	RQ3: The Impact of Hashing	37
5.3.1	Alternative Hashing Algorithms	38
5.4	RQ4: The Impact of Tracing Memoized Data	39
6	Conclusion	44
6.1	Future Work	44
6.2	Software	45
	Bibliography	46
	Appendix A Object-Oriented Benchmarks	49
	Appendix B Memoization Model in Alloy	55

Chapter 1

Introduction

Efficiency and performance are perpetual concerns in the realm of programming languages. This dissertation delves into a relatively underexplored area—applying memoization to mutable objects—to enhance the efficiency of programs. While memoization has proven effective in optimizing pure functions, its application to mutable state remains a challenge. This research aims to fill this gap by investigating the implications of memoizing mutable state. Motivated by the central role objects play in modern programming, we recognize the need to adapt traditional memoization approaches to their dynamic nature. This exploration seeks to uncover the nuances, address challenges, and potentially extend the usefulness of this old-time technique.

Traditional Memoization Traditional memoization, commonly used in functional programming languages, is a technique designed to optimize the execution of functions by caching their results. The idea is to store the results of expensive function calls and return the cached result when the same inputs occur again—instead of recomputing the result. Memoization finds its application chiefly in functions characterized by *referential transparency*. Søndergaard and Sestoft [24] define referential transparency as the quality exhibited by a function which, in every instance of its execution, given the same input, yields an identical result. This property proves indispensable for memoization since subsequent calls to a memoized function after its initial invocation are substituted with cached data.

The necessity for immutable values stems from the potential alterations to cached values. Mutable values could trigger alterations spanning across all references to the cached value. Consequently, memoization rarely integrates into the framework of object-oriented programs. Objects inherently harbor mutable states, thereby risking the use of invalid data across analogous objects constructed with identical parameters. Section 2.2 illustrates how the memoization of object instantiation routines could harm the semantics of Python programs.

Modern programming languages provide mechanisms to facilitate the application of memoization, exemplified by constructs such as `@functools.cache` or `@functools.lru_cache` in Python [28], the `#[cached]` macro within the `cached` [16] library for Rust, and the `memoFix` function wrapper from the `Data.Function.Memoize` library for Haskell [7]. Listing 1.1 provides an example of their usage. These tools facilitate the process of implementing memoization for regular functions; however, their usage demands caution due to potential side-effects inherent in certain languages (such as Python and Rust).

The Contributions of This Work This research introduces a technique that alleviates the strictness of the immutability requirement, thus enabling the memoization of routines that yield mutable values. This approach combines *shared-ownership pointers* [31] with

<pre> 1 from functools import cache 2 3 @cache 4 def fib(n): 5 if n == 0 or n == 1: 6 return n 7 return fib(n-1) + fib(n-2) </pre> (a) Python	<pre> 1 use cached::proc_macro::cached; 2 3 #[cached] 4 fn fib(n: u64) -> u64 { 5 if n == 0 n == 1 { return n 6 ↪ } 7 fib(n-1) + fib(n-2) 8 } </pre> (b) Rust
<pre> 1 import Data.Function.Memoize 2 type Memo f = f -> f 3 4 fib :: Memo (Int -> Integer) 5 fib fib 0 = 0 6 fib fib 1 = 1 7 fib fib n = fib (n - 1) + fib (n - 8 ↪ 2) 9 10 fibMemo :: Int -> Integer 11 fibMemo = memoFix fib </pre> (c) Haskell	

Listing 1.1: Examples of memoization code to compute Fibonacci numbers in different programming languages.

the *copy-on-write* [6] policy. Detailed implementation specifics are deferred to Chapter 3; however, in essence, our memoization implementation guarantees the following sequence of properties—where o is an object, and r is a reference that points to o .

1. If an object is memoized, all states of the program remain valid.
2. If an object is created in the memoization table, then it remains in the memoization table unless modified.
3. If a pointer r is utilized to access an operation mutating o , r then points to a non-memoized copy of o before mutation.
4. When o is memoized, any property within o can be accessed via r with two pointer dereferences (i.e., two load operations).
5. If o is not memoized, accesses to o via r incur no performance penalty, requiring only one load operation.

The most relevant properties are 1 and 2, as they ensure we have safe memoization. Chapter 4 formalizes the concept of memoized objects and verify these guarantees on the Alloy [13] specification language.

Summary of Results The proposed methodology has been implemented in the Hush programming language [5], a scripting language similar to Lua. Section 2.3 outlines key

features of Hush relevant to this study. Hush was chosen due to its support for object-oriented programming by creating objects as closures. By memoizing routines yielding such closures, we effectively memoize the ubiquitous `new` operation in object-oriented programming languages. Experiments detailed in Chapter 5 show that our memoization implementation experiences only marginal slowdown compared to unsafe memoization (as exemplified in Python, for instance, and discussed in Section 2.2). Furthermore, depending on the specific problem, safe memoization grants all the advantages of traditional memoization—originally tailored for immutable values—onto object-oriented environments where objects encapsulate mutable state. Incorporating safe memoization into object creation routines brings in a notable outcome: it makes the *flyweight* design pattern [8] native to these languages.

This dissertation comprises six chapters, inclusive of this introduction. Chapter 2 provides an overview of prior memoization work, alongside necessary background concepts. Chapter 3 explains various memoization approaches developed within the Hush interpreter, aimed at addressing the challenge of mutable memoization. Chapter 4 formalizes the concepts introduced in Chapter 3 and verifies their properties using the Alloy [14] specification language. Chapter 5 addresses research questions regarding the performance of the techniques devised throughout this study. Finally, Chapter 6 offers concluding thoughts, including the limitations of our approach and potential avenues for future research. Additionally, the benchmark programs assessed in Chapter 5 are provided in the appendix for reference.

Chapter 2

Background

Memoization is a well-known programming technique. This technique consists in caching the results of idempotent functions, so that new invocations of said functions avoid recomputing values already seen. Typically, memoization is implemented by the programmer, as a design pattern. However, there are many different attempts to add memoization as an automatic feature to different programming languages. Section 2.1 shall explore some of these techniques. In this work, we shall implement automatic memoization in the Hush programming language. Given that this language has a relatively small user base, Section 2.3 shall revisit some details of this language’s implementation. Finally, Section 2.4 will provide a brief overview of the formal techniques that Chapter 4 uses to formalize the contents of the present work.

2.1 Automatic Memoization

The idea of “memo” functions was introduced by Donald Michie in 1968 [19]. Since then, memoization has been used in many different contexts and at various levels of granularity—most commonly at the granularity of a function, however. Implementations range from software-based [2, 20, 22, 23, 25, 26], hardware-based [33], or a combination of both [18, 29]. In this context, there are two steps to perform memoization effectively: (i) identify which portions of the code are suitable for memoization, and may profit from it; (ii) transform the identified code into a variant that is able to memoize, thus storing results and retrieving it in case the same input occurs. In most cases, both steps are performed by the programmer, such as in the competitive programming scene [27]. If the second step is performed programatically, it is generally called *automatic memoization* [20].

In their work, Suresh et al. [30] devised a load-time technique to identify a set of computationally intensive pure functions—referred to as transcendental functions—and apply a memoization wrapper to them. This approach yielded performance improvements of up to 50% on Intel Ivy Bridge processors and up to 10% on ARM Cortex-A9 processors. Initially applicable only to dynamically linked functions from external libraries, this technique was later extended in 2017 [29] to a compile-time analysis capable of intercepting user-defined and statically linked functions. By optimizing the handling of constants and leveraging the inlining capability of the LLVM compiler, this extension also delivered notable performance gains. Nonetheless, their technique targets pure functions exclusively, utilizing global variables and pointers as criteria for function selection.

In addition to managing mutability, another primary concern to implementing memoization is the table look-up time. A fast look-up time minimizes overhead introduced by memoization code. Ideally, the look-up time should not exceed the time to execute the original computation, as this would lead to a performance regression. Over time, various

methods have been proposed to speed up this search process. For instance, approximate computing has been used to search the memoization table for similar matches rather than exact ones [3], although such techniques often result in a degradation of program output. Alternatively, other approaches capitalize on the temporal or spatial locality of data for computation reuse [32, 33]. Kamimura et al. [15] address this challenge by partitioning the table into smaller units and constraining their sizes.

The cited work in this section are examples techniques of memoizing referentially transparent functions. Therefore, mutability poses a problem to the effective implementation of their techniques. Our work deals with this limitation by employing a different approach to memoization based on the copy-on-write [6] policy. Doing this we are able to memoize values which can later be modified. The next section shall present this problem and motivate it by showing a concrete example.

2.2 The Challenge of Mutability

Memoizing mutable values can introduce the risk of designing incorrect programs due to the creation of unintended *aliases*. Aliasing, a characteristic of imperative programming languages, enables multiple pointers to reference the same memory location. Example 2.1 illustrates how memoizing mutable objects may carelessly lead to the implementation of incorrect programs.

Example 2.1. Listing 2.1 shows a Python program that implements bidimensional points. Points are instances of the `Point` class. The `@cache` annotation forces memoization of the `__init__` routine that creates objects. Thus, objects created with the same initializing parameters are memoized. Consequently, references `p0` and `p1` point to the same object in the memoization table. When a property of `p1` is modified, the same property of `p0` is also affected. This behavior is erroneous because the program does not include any explicit copy operation (such as `p0 = p1`) that would cause these objects to alias.

```
1 from functools import cache
2
3 @cache
4 class Point:
5     def __init__(x, y):
6         self.x = x
7         self.y = y
8
9 p0 = Point(1, 1)
10 p1 = Point(1, 1)
11
12 p1.x = 2
13
14 assert(p1.x == 2) // Pass...
15 assert(p0.x == 1) // Fail!
```

Listing 2.1: Example where the memoization of a routine that creates mutable values causes the program to be incorrect.

Incremental computation (IC) is another technique that aims to improve the efficiency of a program by reusing previously computed results. In contrast to memoization,

IC works by updating only the output parts affected by input changes [12]. It is useful for scenarios where inputs change incrementally and recalculating everything would be inefficient. Acar, Ahmed, and Blume [1] proposed *change propagation* to address mutability in the context of IC. They also introduced a data structure called *dynamic dependency graph* (DDG) to track which values might need to be recomputed due to side-effects. Combining DDGs to track changes and applying memoization selectively is the essence of self-adjusting computation [1, 12].

One drawback of this technique is recomputing all dependencies of a changed value, even if they are no longer required—not reachable from a particular point forward. Another issue is the potential overhead introduced from constantly updating the DDG in functions that are rarely called with similar inputs. Hammer et al. [11] addressed these problems with Adapton, introducing a demand-driven semantics to change propagation, in a lazy evaluation style. Along with a new demand-driven change propagation algorithm, they also introduced the *demand computation graph* (DCG) data structure. Although correct, these techniques require very specialized libraries for IC and do not offer a ready-to-use mechanism for programmers, such as a language keyword.

Example 2.2. Listing 2.2 shows a Rust program that uses memoization through macros provided by Adapton [11]. Similar to Example 2.1, this program creates memoized points, however, in this case both `p` and `q` remain unmodified. We opted for a function that return instances of points, for this function is easier to memoize than the typical constructor for structs in Rust. The verbosity of this program—apart from language choice—arises from the greater flexibility of this framework; it permits different levels of granularity on which to apply memoization to. The macro `memo!` executes a memoized version of the `create_point` function, passing the same arguments in both cases, therefore avoiding a second call to the function. The assert in line 21 passes, the one in line 22 fails. We conclude that even though `p` and `q` have the same values, they point to different objects. These objects could be safely reused in this program, but Adapton unnecessarily copies `p` at the second call to `create_point`.

```

1  #[macro_use] extern crate adapton;
2  use adapton::macros::*;
3  use adapton::engine::*;
4
5  #[derive(PartialEq, Eq, Clone, Hash, Debug)]
6  struct Point {
7      x: i32,
8      y: i32
9  }
10
11 fn create_point(x: i32, y: i32) -> Point {
12     Point { x, y }
13 }
14
15 fn main() {
16     manage::init_dcg();
17
18     let (ap, p): (Art<Point>, Point) = memo!([Some(name_unit())]? create_point;
        ↪ x:1, y:1);

```

```

19     let (aq, q): (Art<Point>, Point) = memo!([Some(name_unit())]? create_point;
    ↪ x:1, y:1);
20
21     assert_eq!(p, q); // Pass...
22     assert_eq!(std::ptr::addr_of!(p), std::ptr::addr_of!(q)); // Fail!
23 }

```

Listing 2.2: A Rust program similar to Listing 2.1, but leaving the points `p` and `q` unmodified. Line 16 initializes the demanded computational graph (DCG) used to track changes.

Rito and Cachopo [23] were the first to adapt memoization to the characteristics of object-oriented languages. They used Software Transactional Memory (STM) to track internal state dependencies and side-effects in Java. The STM system manages references to shared mutable state by treating operations on memoized values as *transactions* that can be aborted if there are conflicting changes. However, implementing their technique requires an available STM system for the target language. Another drawback is the possible performance hit from maintaining per-transaction logs and committing changes. In contrast, our approach employ smart pointers and copy-on-write for managing mutability (see Section 3.1). This method is simpler to implement and only slightly increases memory usage compared to traditional memoization (see Section 5.2).

2.3 The Hush Programming Language

We chose the Hush programming language to demonstrate the ideas discussed in this document. Hush is a dynamically typed scripting language, conceived to orchestrate the execution of commands on an operating system’s shell. Our choice of Hush was motivated by a single characteristic of that language: it creates objects as function closures. Thus, by memoizing such functions, we automatically obtain memoized mutable objects. Listing 2.3 shows how objects are created in this programming language.

```

1 let Counter = function(start)
2   @[
3     _start: start,
4
5     get: function()
6       self._start
7     end,
8
9     increment: function()
10      self._start = self.start + 1
11    end,
12  ]
13 end

```

Listing 2.3: Function that fills the role of a class in Hush.

Hush’s runtime is implemented in Rust. This runtime is slower than Python’s, for instance, however, not by a large margin. Listings 2.4 and 2.5 shows two programs: one written in Hush, and another version of it written in Python. These programs implement a solution to the classic Knapsack problem; with time complexity of $O(nC)$, where n is the number of different items, and C is the total capacity of the knapsack. The solution

uses dynamic programming to memoize subproblems, and thus compose the solution to each problem [9]. As the reader will notice, these languages offer similar syntax.

<pre> 1 std.randseed(0) 2 3 let C = std.int(std.args()[0]) 4 let N = std.int(std.args()[1]) 5 6 let v = [] 7 let w = [] 8 9 function max(a, b) 10 if a >= b then a 11 else b 12 end 13 end 14 15 memo function knapsack(c, n) 16 if c == 0 or n == 0 then 17 0 18 elseif w[n-1] > c then 19 knapsack(c, n-1) 20 else 21 max(v[n-1] + knapsack(c-w[n-1], n-1) 22 ↪ , knapsack(c, n-1)) 23 end 24 end 25 26 for _ in std.range(0, N, 1) do 27 std.push(v, std.randint(0, 100)) 28 std.push(w, std.randint(0, 100)) 29 end 30 std.println(knapsack(C, N)) </pre>	<pre> 1 import sys 2 import random 3 from functools import cache 4 5 random.seed(0) 6 7 C = int(sys.argv[1]) 8 N = int(sys.argv[2]) 9 10 v = [] 11 w = [] 12 13 @cache 14 def knapsack(c, n): 15 if c == 0 or n == 0: 16 return 0 17 elif w[n-1] > c: 18 return knapsack(c, n-1) 19 else: 20 return max(v[n-1] + knapsack(c- 21 ↪ w[n-1], n-1), knapsack(c, 22 ↪ n-1)) 23 24 for _ in range(N): 25 v.append(random.randint(0, 100)) 26 w.append(random.randint(0, 100)) 27 28 print(knapsack(C, N)) </pre>
---	---

Listing 2.4: Hush program

Listing 2.5: Python program

Figure 2.1, in turn, shows a comparison between the runtime performance of these two programs. In this example, the Hush program is seven times slower than the equivalent Python program for the largest input. Hush tends to be slower, not as much due to any fundamental characteristic of the language. Rather, its runtime subsumes within its implementation less engineering hours than the CPython virtual machine. Nevertheless, this environment shows gives us the subsidies to demonstrate how mutable objects can be correctly and efficiently memoized.

2.4 Theorem Provers and Model Checkers

This section introduces basic concepts of formal specification and motivates the choices made in Chapter 4. Our goal is not to rigorously explain the inner workings of the tools used but to clarify and define our ideas formally. We use the Alloy specification language to achieve this goal [13]. Alloy was developed for modeling and verifying properties using

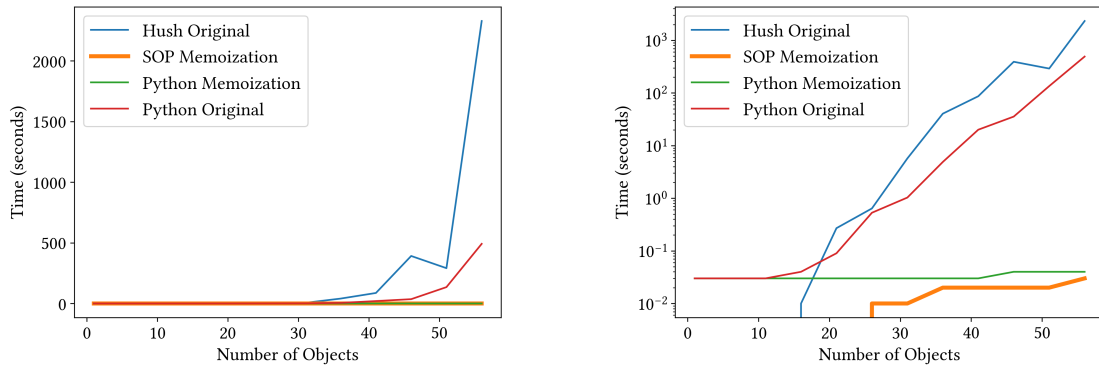


Figure 2.1: Execution time of *knapsack* on different y-axis scales.

a combination of the quantifiers from first-order logic with set-theoretic and relational calculus operators. It is a tool that incorporates aspects of both theorem proving and model checking but leans more towards model checking. The next paragraphs shall elaborate these concepts and make this distinction clear.

Theorem provers are crucial in computer science for ensuring the correctness of software, hardware, and mathematical proofs. They form the foundation for formal verification and automated reasoning. Theorem provers fall into two categories: automated (ATP) and interactive (ITP). ATPs, such as Z3 and cvc5, attempt to find proofs automatically. ITPs, like Coq, Isabelle/HOL, and Lean, require human guidance to construct proofs.

Model checkers are tools used to automatically verify the correctness of finite-state systems. They systematically explore the state space of a system to check whether a given specification, usually expressed in temporal logic, holds. It capitalizes on the idea of finding all small counter-examples. Such approach relies on the *small scope hypothesis*, which states that most bugs can be found with a small quantity of entities. The Alloy Analyzer allows the programmer to define the scope of each structure in the model. Doing so is important for keeping reasonable running times for its SAT solver.

Transition Systems The dynamic aspects of a system, such as its behavior over time, are often modeled using a combination of state and transition relations. A transition system can be represented as a sequence of states and transitions between these states. Each state is a snapshot of the system at a specific point in time. In this context, a *trace* is a sequence of states that fully describe the behavior of a system. In Alloy, transitions are defined implicitly by using temporal logic formulas, an idea proposed by Leslie Lamport in 1994 [17]. The Alloy Analyzer explores the state space of the model systematically to check whether the specified properties hold over different traces.

To verify properties like “every object that is reachable will not be garbage collected” and “memoized objects are always reachable”, we create an Alloy model that represents the system. This model includes components such as objects, references, garbage collection behavior, memoization mechanisms, and the rules governing reachability. The overall approach works as follows:

1. Define your domain: Model the objects in your system, including memoized objects and other objects that may be involved in the reachability analysis.

2. Define relationships: Specify the relationships between objects, such as references between them and how memoization affects these relationships.
3. Specify garbage collection behavior: Describe the rules for garbage collection in your system, including when objects are considered unreachable and eligible for garbage collection.
4. Formulate assertions: Express your properties as assertions in Alloy. For example, you could assert that for every object that is reachable, it will not be garbage collected. Similarly, you could assert that all memoized objects are reachable.
5. Analyze the model: Use the Alloy Analyzer to check if your assertions hold true for all possible states of the model. If they do, then your properties are verified; otherwise, the analyzer may provide counterexamples to help you understand why the properties fail.

Alloy offers several advantages for verifying properties of software and system designs compared to other formal verification tools like Coq or Z3. Firstly, Alloy provides a higher-level, more declarative language compared to tools like Coq or Z3. Its syntax is designed to be intuitive and easy to understand, which can make modeling complex systems more manageable, especially for those who are not experts in formal methods. Secondly, Alloy comes with an automatic analysis engine that explores the state space of your model to check the specified properties. This automatic analysis can quickly identify counterexamples or confirm the validity of properties without requiring manual proof construction or exploration. While Coq and Z3 are more powerful and can handle larger, more complex systems, they often require significant manual effort to construct proofs or encode properties.

Chapter 3

Shared Ownership of Memoized Values

This chapter explains our implementation of memoization for mutable values. Our approach requires modifications to the language runtime system and cannot be implemented as a library. This necessity arises because we change how properties of memoized objects are accessed. In our approach, accessing these properties requires an extra pointer dereference, in addition to the normal property lookup. This chapter is organized as follows. Section 3.1, provides a brief overview of *shared-ownership pointers*, and how we them in our memoization technique. Section 3.2 explains the different ways objects are created using shared pointers. Section 3.3 describes how objects are accessed depending on how they were created. Lastly, Section 3.4 justifies the use of *copy-on-write* to preserve the semantics of programs.

3.1 Shared-Ownership Pointers

Shared-ownership pointers are a concept often used in programming languages like C++ and Rust, as Example 3.1 explains. This abstraction prevents memory leaks. A shared-ownership pointer is a pointer that keeps track of how many references (or “owners”) there are to a particular object.

Example 3.1. In C++, `std::shared_ptr` is a commonly used shared-ownership pointer implementation provided by the standard library. In Rust, an equivalent abstraction could be achieved using the `Rc<T>` (Reference Counted) smart pointer provided by the standard library. `Rc<T>` is used for immutable data. To share ownership of mutable values, Rust programmers can use `RefCell<T>` to make use of a pattern called *interior mutability*.

In this work, shared-ownership pointers are not a resource to avoid memory leaks. Rather, they are a means to implement the “copy-on-write” policy of memoized objects. We have modified the Hush runtime environment to only refer to memoized objects through shared pointers. However, non-memoized objects are still accessed through plain references. Figure 3.1 illustrates the three key operations that our runtime environment performs on objects. In what follows, these operations refer to one of the two programs in Figure 3.1 (a):

- As seen in Figure 3.1 (b), if $c0 = f()$ is a reference to an object o allocated by function f , then $c0$ points to o on the heap, but outside the memoization table.
- As seen in Figure 3.1 (c), if $p0 = g()$ is a reference to an object o created by a memoized function g , then $p0$ holds the address of a shared pointer `sop0`, and `sop0` holds the address of o , which is stored in the memoization table.

- As seen in Figure 3.1 (d), if $p0 = sop$ is a reference to an object o , and o is updated, then o is copied into a new object o' , which is outside the memoization table. The shared reference sop is updated to point to o' .

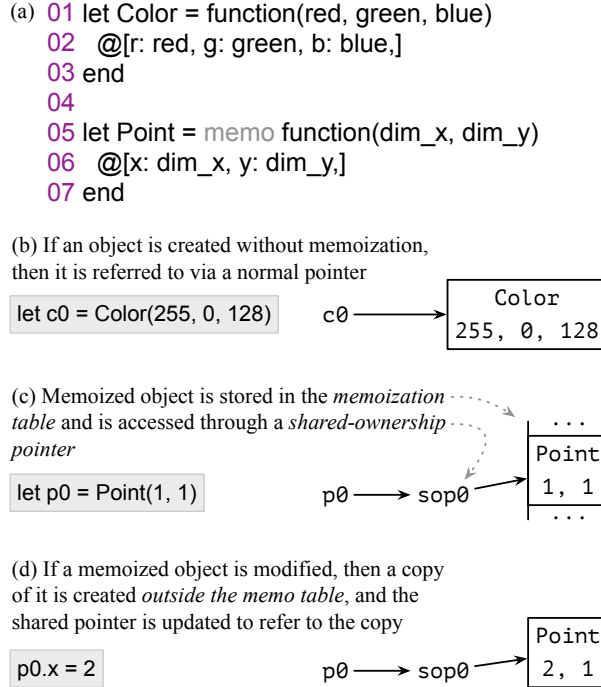


Figure 3.1: Distinction between non-memoized and memoized objects. The latter are accessed through a shared pointer, and stored in the memoization table unless modified.

If two pointers refer to the same memory location, then we say that they are *aliases*. In this work, we use shared-ownership pointers with the purpose of distinguishing between three kinds of aliases:

True: These are the typical aliases created through assignment between non-memoized objects. If r_0 is a reference to a non-memoized object o , then an assignment such as $r_1 = r_0$ forces r_1 to be an *alias* of r_0 , meaning that these two references hold the same address; namely, the address of o in memory.

Logical: These are aliases created through assignments where the right side is a memoized object. If r_0 is a reference to a memoized object o , then an assignment such as $r_1 = r_0$ forces r_1 to be an *alias* of r_0 . In this case the value of r_0 equals the value of r_1 , which is the shared-ownership pointer originally pointed to by r_0 .

Memoized: These are “unintentional” aliases created due to memoization. Let $r_0 = f(i)$ and $r_1 = f(i)$ be two references to an object created by two calls of the memoized function f , with the same argument i . References r_0 and r_1 refer to the same memory location, which exists within the memoization table; thus, they are *aliases*. However, $r_0 \neq r_1$, as they point to different shared-ownership pointers.

Notice that in true and logical aliasing, the values of the references r_0 and r_1 are the same. However, in the former case, they point directly to the object o , whereas in the latter, they point to a shared pointer. In contrast, in the case of memoized aliases, the values in the references differ: they point to different shared pointers, but these shared

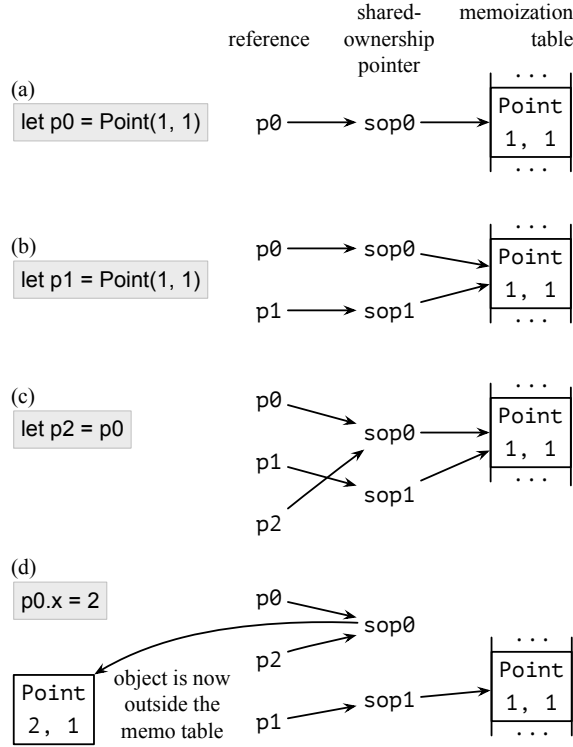


Figure 3.2: (a) Creation of memoized object. (b) Memoized aliases. (c) Logical aliases. (d) Copy-on-write semantics.

pointers refer to the same object in the memoization table. Example 3.2 shall make this distinction clearer.

Example 3.2. Figure 3.2 shows some operations performed onto memoized instances of function `Point` (seen in Figure 3.1). As seen in Figure 3.2 (a), memoized objects are accessed via shared pointers. Figure 3.2 (b) shows that the construction of two instances of the same memoized object leads to the creation of two distinct shared pointers. In this case, `p0` and `p1` are *memoization aliases*. Notice that `p0` and `p1` are not aliases in the strict sense: they refer to different shared pointers; hence, `p0` $\neq$ `p1`. However, assignments, such as the one shown in Figure 3.2 (c) lead to the creation of logical aliases. These assignments do not create new shared pointers: the aliases point to the same shared-ownership reference; thus, they contain the same address, e.g.: `p0` = `p1`. Therefore, if any of these aliases is used to modify an object, they all are automatically updated, once the shared pointer is reassigned to a copy of the modified object, as Figure 3.2 (d) shows.

3.2 Object Creation

Our implementation of Hush creates objects in one of three ways, which Figure 3.3 subsumes via a simplified set of operational rules. The first rule shows the creation of non-memoized objects. In this case, objects are created as in other languages: `self` $\leftarrow$ `new(i)` allocates memory to a new object, making `self` a pointer to this newly created location.

Creation of objects via memoized functions, e.g., as due to `self` $\leftarrow$ `memo new(i)`, always leads to the creation of a fresh shared pointer `s`. However, from this point on, object creation follows one of two possibilities. If the memoization table `t` does not contains an

$$\begin{array}{c}
\frac{o = \text{new}(i)}{\text{env} \vdash \text{self} \leftarrow \text{new}(i) \text{ is env}[\text{self}] \leftarrow o} \\
\\
\frac{t = \text{env}[\text{memo}] \quad t[i] = \perp \quad s = \text{fresh} \quad o = \text{new}(i)}{\text{env} \vdash \text{self} \leftarrow \text{memo new}(i) \text{ is env}[\text{self}] \leftarrow s} \\
\quad \text{and env}[s] \leftarrow o \text{ and } t[i] = o \\
\quad \text{and memo} \supset \{\text{self}\} \\
\\
\frac{t = \text{env}[\text{memo}] \quad t[i] = o \quad s = \text{fresh}}{\text{env} \vdash \text{self} \leftarrow \text{memo new}(i) \text{ is env}[\text{self}] \leftarrow s \text{ and env}[s] \leftarrow o} \\
\quad \text{and memo} \supset \{\text{self}\}
\end{array}$$

Figure 3.3: Rules for creating objects. The environment “env” associates variable names with values.

entry for i , then a new block of memory o is allocated and stored in t , as seen in the second rule of Figure 3.3. Otherwise, the object o stored at $t[i]$ is returned. In both cases, the new shared pointer s will refer to o .

3.3 Property Accesses

Our implementation of Hush’s code generator performs object access in two different ways, as Figure 3.4 shows. Properties of non-memoized objects are retrieved via a search in the object table, just like in other dynamically-typed programming languages such as Python or Ruby. The first rule in Figure 3.4 shows this operation.

$$\begin{array}{c}
\frac{o = \text{env}[\text{self}] \quad f = o[m]}{\text{env} \vdash \text{self.m}(i) \text{ is } f(\text{self}, i)} \quad \text{self} \notin \text{memo} \\
\\
\frac{s = \text{env}[\text{self}] \quad o = \text{env}[s] \quad f = o[m]}{\text{env} \vdash \text{self.m}(i) \text{ is } f(\text{self}, i)} \quad \text{self} \in \text{memo}
\end{array}$$

Figure 3.4: Rules to access object properties.

Properties of objects created by memoized functions are accessed through the shared pointer, as seen in the second Rule of Figure 3.4. Notice that, as per these two rules, different sets of instructions are generated to implement objects, depending on how they were created—if via memoization or not. Figure 3.3 show these code generation rules.

3.4 The Copy-On-Write Semantics

According to the first rule in Figure 3.5, the modification of non-memoized objects happens as in any other dynamically typed language. In this case, the memory o pointed out by a reference **self** is retrieved from the environment, and the property of interest within o is updated.

However, the modification of a memoized object o causes a copy o' of this object to be created. This process is represented by the second rule in Figure 3.5. There are three important facts that must be mentioned:

$$\begin{array}{c}
\frac{o = \text{env}[\mathbf{self}]}{\text{env} \vdash \mathbf{self.p} \leftarrow x \text{ is } o[p] \leftarrow x} \quad \mathbf{self} \notin \text{memo} \\
\\
\frac{s = \text{env}[\mathbf{self}] \quad o = \text{env}[s] \quad o' = \text{copy}[m]}{\text{env} \vdash \mathbf{self.p} \leftarrow x \text{ is } o'[p] \leftarrow x \text{ and } \text{env}[s] \leftarrow o'} \quad \mathbf{self} \in \text{memo}
\end{array}$$

Figure 3.5: Rules to modify object properties.

- The copy o' is not stored in the memoization table.
- The original version of object o is not removed from the memoization table; thus, remaining memoized.
- The new copy o' remains accessed via the old shared-ownership pointer s ; hence, logical aliases of the modified pointer $\mathbf{self}$ do not need to be updated.

Chapter 4

Formal Specification

This chapter provides a model of memoization using the Alloy specification language [14]. This model should specify our general technique for memoizing mutable objects and verify its soundness. Therefore, our Alloy code shall elicit the necessary components for representing the semantics of the concrete implementation, without oversimplifying it. In Section 4.1, we provide a brief introduction to the language; readers proficient in Alloy may skip this section. Lastly, Section 4.2 introduces the memoization model, along with core definitions and properties to be verified. The full specification of this model is available in Appendix B.

4.1 Introduction to Alloy

Alloy is a declarative specification language used to design and analyze models of complex systems. It focuses on expressing structural constraints and behavior in a clear and concise way. Alloy specifications are based on first-order logic and set operations, making them formally precise and unambiguous. This enables automated analysis and verification of models by using the Alloy Analyzer. Its latest version (Alloy 6) adds support mutable relations and linear time temporal logic, making it easier to reason about dynamic systems.

Alloy works by reasoning about sets and relations. The way to define a set is through a signature by using the keyword **sig**, followed by a required name. Additionally, a signature might have a number of *fields* enclosed between braces. Fields are relations combining its signature to other elements of the model. The elements contained in sets are named *atoms*; they have no internal semantics and are generated by the Alloy Analyzer. Listing 4.1 provides some signatures from the model in Section 4.2.

```
abstract sig ObjectStatus {}
one sig Reachable, Unreachable extends ObjectStatus {}

var sig Object {
  var status: one ObjectStatus,
  var points: set Object
}

var sig Params {
  var memo: one Object
}

var sig Reference {
  var alias: set Reference,
  var points: lone Object
}
```

Listing 4.1: Signatures in the model.

The keyword **abstract** means that every signature that extends it defines a partition on that set. Therefore, every element of **ObjectStatus** must be in one of its subsets

Reachable or **Unreachable**. Further, given the keyword **one**, each partition must be a set with exactly one element. We also define the relations *status* and *points* on **Object**, where the multiplicity **lone** means “at most one”. In this example, therefore, objects can have the *status* of reachable, unreachable, or none whatsoever, and *points* to other objects. Further, unless indicated with the keyword **var**, everything is immutable by default. Given the signatures defined in Listing 4.1, we are now able to find instances that satisfy this model.

Example 4.1. Figure 4.1 shows a graphical representation of an instance found by the Alloy Analyzer with the signatures from Listing 4.1. The boxes represent elements of sets and arrows how they are related. In this example, the elements **Params0** and **Object0** of sets **Params** and **Object**, respectively, are related by the **memo** relation. The singleton **Track** indicates which operation was performed in the last state transition. Therefore, the current state instantiate an object with the **NEW** operation. Other operations of this model shall be presented in Section 4.2

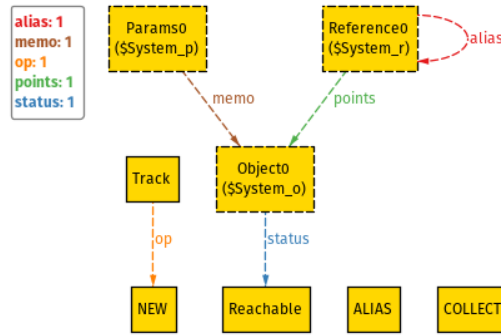


Figure 4.1: Instance found by the Alloy Analyzer.

4.2 The Memoization Model

The Hush programming language features a mark-and-sweep garbage collector. We have formalized some properties that arise from the interaction of memoization and garbage collection. Listing 4.2 states these properties. We use Alloy’s SAT solver to find counter-examples. The rest of this section provides an overview of this process.

The *memo* relation which we denote by M is particularly important, as it maps parameters to objects, effectively acting as the memoization table. To illustrate this, let O be the set of objects, let P the set of parameters, and R the set of references in the program, then $M = \{(p, o) \in P \times O : new(r, p, o) \text{ for some } r \in R\}$ where the predicate *new* is defined as follows.

$$new(r, p, o) \equiv \begin{cases} \text{true} & \text{if } o \text{ is an object allocated with parameters } p \text{ and } r \text{ points to } o. \\ \text{false} & \text{otherwise} \end{cases}$$

In languages with automated memory management, such as garbage-collected languages (e.g., Java, Python, C#), a reachable object is one that can be accessed through a chain of references starting from a set of root references. To better define the properties regarding garbage collection in our model, we provide a more precise definition of a reachable object.

Definition 4.1. *An object o is **reachable** if:*

1. *There is a reference $r \in R$ that points to o , or*
2. *There is a reachable object o' that points to o .*

This definition captures the idea that an object is reachable precisely if it can be accessed through references. It considers both direct references and indirect references through other reachable objects. Objects that are not reachable are those that cannot be accessed or reached through any chain of references in the program.

The Hush programming language has, naturally, mechanisms for dealing with objects. These include: allocating, accessing, and changing them. In our model, transitions between states are provoked by one of the following operations:

New: instantiate a new memoized object.

Read: read the state of an object via a simple reference or a shared-ownership pointer.

Write: modify the state of an object following the first rule in Figure 3.5. This operation is unsafe if performed upon memoized objects, as seen in Section 2.2. This operation modifies logical and memoized aliases, as hinted by Property **P4** in Listing 4.2.

Drop: removes a pointer from scope. The pointed-to object might become unreachable. It will be eventually collected, as per Property **P6** in Listing 4.2.

Copy-on-Write: copy and modify the state of an object, following the second rule in Figure 3.5. This operation breaks memoized aliasing relations, as hinted by Property **P5** in Figure 4.2.

Alias: creates an aliasing relation between two pointers, due to an assignment such as $r_0 \ r_1$. If the objects are memoized, then a logical aliasing relation is created; otherwise, a true relation is created.

Although objects might suffer an infinite number of such operations throughout their lifetimes, it is possible to define a restricted set of effects upon them, such that the total number of state configurations reached by any trace becomes finite. In our model, the only operations that provoke effects upon objects are **Write** and **Copy-on-Write**. This effect is a unique “modified” next state, which is restricted enough to ensure termination of our Alloy demonstrations.

4.2.1 Properties

We propose six properties which must be satisfied by our model, aiming at showing that our implementation of memoization is sound.

Property 1 References can only point to objects that are reachable.

Property 2 Objects stored in the memoization table are always reachable.

Property 3 Two objects created via a memoized function with the same parameters are the same, until one of them is modified.

Property 4 Two objects created via a memoized function with the same parameters remain the same if one of them suffers an unsafe write.

Property 5 Two objects created via a memoized function with the same parameters are no the same after one of them suffers a copy-on-write.

Property 6 Every unreachable object will be evetually collected.

```

/* References can only point to objects that are reachable. */
pred p1 [] {
  all r: Reference | one r.points => r.points.status = Reachable
}

/* Objects stored in the memoization table are always reachable.
   For any binary relation r, the domain and range of r
   is r.univ and univ.r, respectively. */
pred p2 [] {
  all o: univ.(memo) | always o.status = Reachable
}

/* Two objects created via a memoized function with the same parameters are the same, until one
   ↪ of them is modified. */
pred p3 [] {
  some disj s0, s1: Reference | some p: Params | some disj o0, o1: Object | {
    new[s0, p, o0] and (not write[s0, o0] until new[s1, p, o1]) => o0 = o1
  }
}

/* Two objects created via a memoized function with the same parameters remain the same if one
   ↪ of them suffers an unsafe write. */
pred p4 [] {
  some s0, s1: Reference | some p: Params | some o0, o1: Object | {
    new[s0, p, o0] and write[s0, o0] and new[s1, p, o1] => o0 = o1
  }
}

/* Two objects created via a memoized function with the same parameters are no the same after
   ↪ one of them suffers a copy-on-write. */
pred p5 [] {
  some s0, s1: Reference | some p: Params | some o0, o1: Object | {
    new[s0, p, o0] and copy_on_write[s0, o0] and new[s1, p, o1] => o0' != o1'
  }
}

/* Every unreachable object will be evetually collected. */
pred p6 [] {
  all o: Object | o.status = Unreachable => eventually collect[] and no o.status'
}

```

Listing 4.2: Properties for the Alloy Analyzer.

Chapter 5

Empirical Evaluation

This chapter evaluates the different implementations of memoization, in particular we aim to answer the following research questions:

- **RQ1:** What is the impact of traditional function memoization in Hush?
- **RQ2:** What is the impact of object memoization in the running time of Hush programs?
- **RQ3:** What is the impact of hashing on the efficiency of memoization?
- **RQ4:** What is the impact of tracing memoized data?

Experimental Setup The experiments were executed on Void Linux (Kernel 6.5.8, musl libc 1.1.24) featuring an AMD Ryzen 7 5700X 16-Core processor and 16GiB of DDR4 memory.

Benchmarks The suite contains 13 handcrafted Hush programs, of which seven are in functional programming style and six in object-oriented style. All programs are available online through the GitHub repository: <https://github.com/caioraposo/hush-benchmarks>.

Measurement Methodology To get the average running time and memory consumption of each program execution, we used the **time** (Version 1.9) tool from the GNU coreutils. For profiling, we use the **flamegraph** (Version 0.6.5) tool downloaded from the repository at <https://github.com/flamegraph-rs/flamegraph>.

Memoization Approaches We evaluate five implementations of Hush. One of them, called **Sop** (short for “Memoization via Shared-Ownership Pointers”), contains the ideas described in this work. The other approaches are:

Original: the default distribution of Hush, which does not support built-in memoization. This version works as a baseline for runtime speed.

Unsafe: a standard implementation of memoization. It might produce wrong programs due to memoization aliasing, as Section 2.2 explains.

Guarded: an implementation of memoization that guards memoized objects against mutation. An attempt to update a property of a memoized object triggers an exception and aborts the program.

Alias Sets: a simulation of shared-ownership pointers outside Hush’s runtime environment. This implementation, available as a library, represents shared pointers as sets containing logical aliases.

5.1 RQ1: Traditional Function Memoization

This work concerns the implementation of memoization of mutable objects in a programming language. In this case, it is of fundamental importance to demonstrate that this programming language supports efficient implementation of memoization in the traditional sense; that is, when referentially transparent functions that return immutable values are memoized. The goal of this section is to build such an argument. We analyze the behavior of traditional memoization when implemented in Hush, aiming at showing that this implementation is practical. In doing this preliminary study, we intend to demonstrate that our implementation is not a “strawman”: when used in the traditional context, it is as efficient as any practical implementation of function memoization.

Methodology This section analyzes seven benchmarks. These benchmarks implement referentially transparent functions that return immutable values. The benchmarks are described in Listings 5.1 through 5.7. We shall compare the original and the memoized implementation of each benchmark. Memoization, again, is obtained by qualifying each function with the `memo` keyword.

Discussion Figure 5.1 shows the running time of programs by input size. All seven benchmarks gives us performance improvement, and in most cases the effect of memoization is immediate even for small inputs. In programs such as **Fibonacci** (Listing 5.1) and **Knapsack** (Listing 5.2) we were able to improve the time complexity of exponential these algorithms into linear time. In the **Max Subarray Sum** (Listing 5.7) benchmark, however, we observed a less dramatic effect, as shown in Figure 5.2, the quadratic time complexity of this program was enhanced into linear. To further expand on this, we additionally ran this benchmark with input sizes up to 10K. We conclude that memoization and array size of 10K is still faster then its original (without `memo` keyword) with array size of 1000.

```

1 memo function fib(n)
2   if n < 2 then
3     1
4   else
5     fib(n - 1) + fib(n - 2)
6   end
7 end

```

Listing 5.1: Fibonacci: Recursive algorithm for retrieving the n th element of the Fibonacci sequence.

```

1 memo function knapsack(c, n)
2   if c == 0 or n == 0 then
3     0
4   elseif w[n-1] > c then
5     knapsack(c, n-1)
6   else
7     max(
8       v[n-1] + knapsack(c-w[n-1], n
9         ↪ -1),
9     knapsack(c, n-1)
10    )
11  end
12 end

```

Listing 5.2: Knapsack: An implementation of a solution to the classic Knapsack problem.

```
1 memo function lev(a, b)
2   if std.len(b) == 0 then
3     std.len(a)
4   elseif std.len(a) == 0 then
5     std.len(b)
6   elseif a[0] == b[0] then
7     lev(tail(a), tail(b))
8   else
9     1 + min(
10      lev(tail(a), b),
11      lev(a, tail(b)),
12      lev(tail(a), tail(b))
13    )
14  end
15 end
```

Listing**5.3:**

Levenshtein: This benchmark computes the Levenshtein distance, which is a string metric for measuring the difference between two sequences.

```
1 memo function sub(m, n)
2   if (m == 0 and n == 0) or n == 0
3     ↪ then
4     return 1
5   elseif m == 0 then
6     return 0
7   end
8   if a[m - 1] == b[n - 1] then
9     sub(m - 1, n - 1) + sub(m - 1, n
10    ↪ )
11  else
12    sub(m - 1, n)
13  end
14 end
```

Listing 5.4: Substring: Given two strings, this algorithm finds the number of times the second string occurs in the first, whether continuously or discontinuously.

```

1 memo function lambda_terms(n, k)
2   if n <= 0 then
3     return 0
4   end
5   if n == 1 then
6     return k
7   end
8
9   let answer = lambda_terms(n - 5, k
    ↪ + 1)
10  for i in std.range(0, n - 2, 1) do
11    answer = (
12      answer +
13      (
14        lambda_terms(i, k) *
15        lambda_terms((n - 2) - i, k)
16      ) % MOD) % MOD
17  end
18  answer
19 end

```

Listing 5.5: Lambda Count: Calculates the number of distinct closed lambda-terms that can be written using at most n symbols.

```

1 memo function min_path_sum(t, i, j,
    ↪ k)
2   if i == k - 1 then
3     t[i][j]
4   else
5     min(
6       min_path_sum(t, i + 1, j, k),
7       min_path_sum(t, i + 1, j + 1, k
    ↪ )
8     ) + t[i][j]
9   end
10 end

```

Listing 5.6: Triangle: Given a triangle array, return the *minimum path sum* from top to bottom. This implementation runs in $O(n)$ where n is the depth of the triangle.

```

1 memo function mss(v, i)
2   if i == 0 then
3     v[0]
4   else
5     max(mss(v, i-1) + v[i], v[i])
6   end
7 end
8
9 let answer = 0
10 for i in std.range(0, std.len(v),
    ↪ 1) do
11   answer = max(answer, mss(v, i))
12 end

```

Listing 5.7: Max Subarray Sum: Calculates the maximum subarray sum, and uses memoization to run in time complexity $O(n)$ instead of $O(n^2)$, where n is the array length.

5.2 RQ2: Memoization of Objects

This section evaluates the memoization of objects as an optimization technique. The idea is to memoize the constructor of an object so that corresponding calls return the same location in memory. Our goal is to investigate the behavior of programs when instances of objects are reutilized.

Methodology We consider programs which implement functions that return objects. This is the default technique used in Hush to instantiate objects: an object is created as a closure. In terms of measurement methodology, we use the same approach already seen in Section 5.1. Namely, we shall increase the inputs of the target functions, which, in

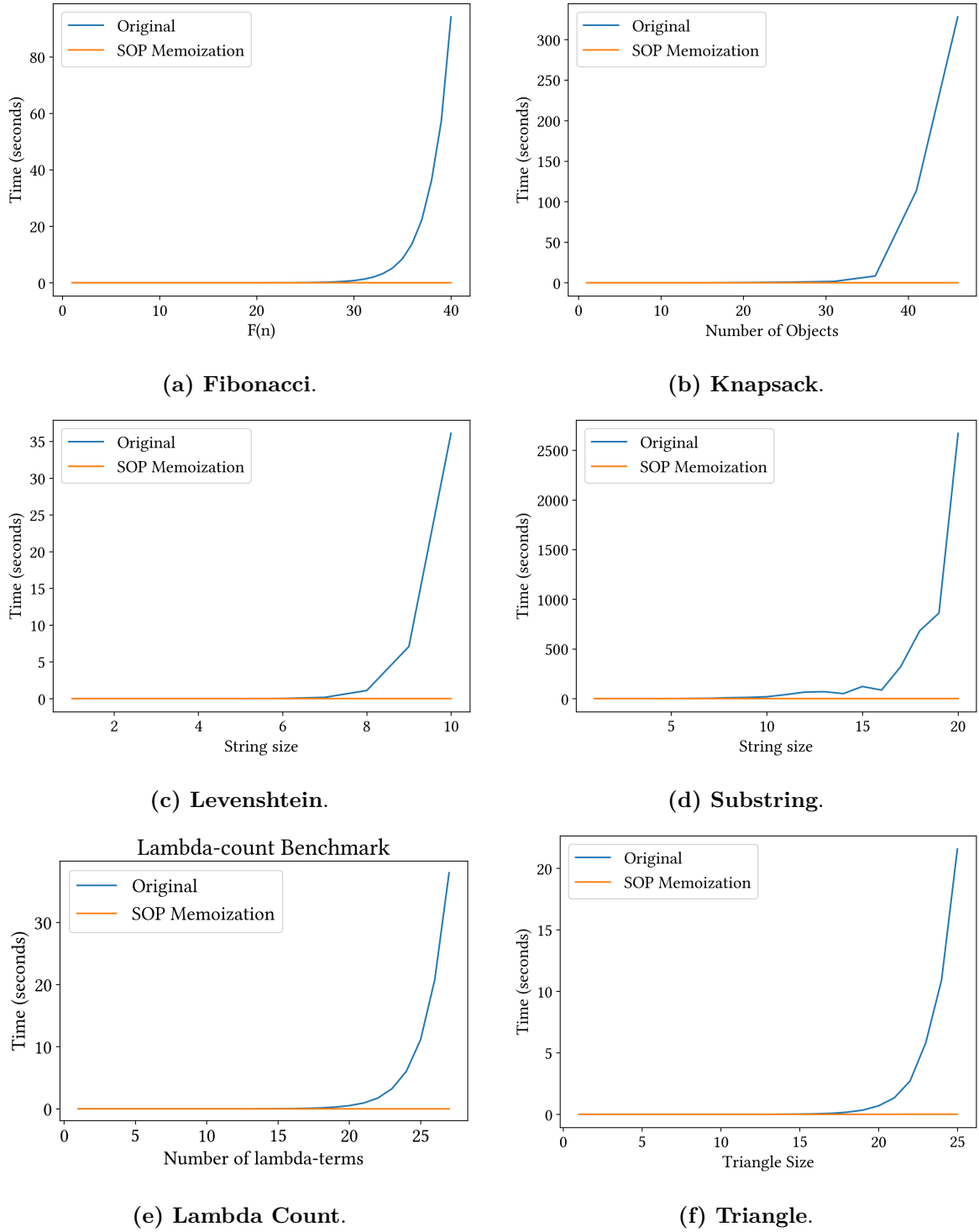


Figure 5.1: Running time of different memoized programs in functional programming style.

turn, shall instantiate a larger number of objects. There is no standard benchmark suite for Hush; therefore, we shall evaluate memoization on six programs of our own craft:

Eval: an evaluator of arithmetic expressions. Expressions are either numbers of variables, or the composition of other expressions, such as addition, multiplication or division of subexpressions.

Figures: an application that represents figures as either primitive shapes (circles, trian-

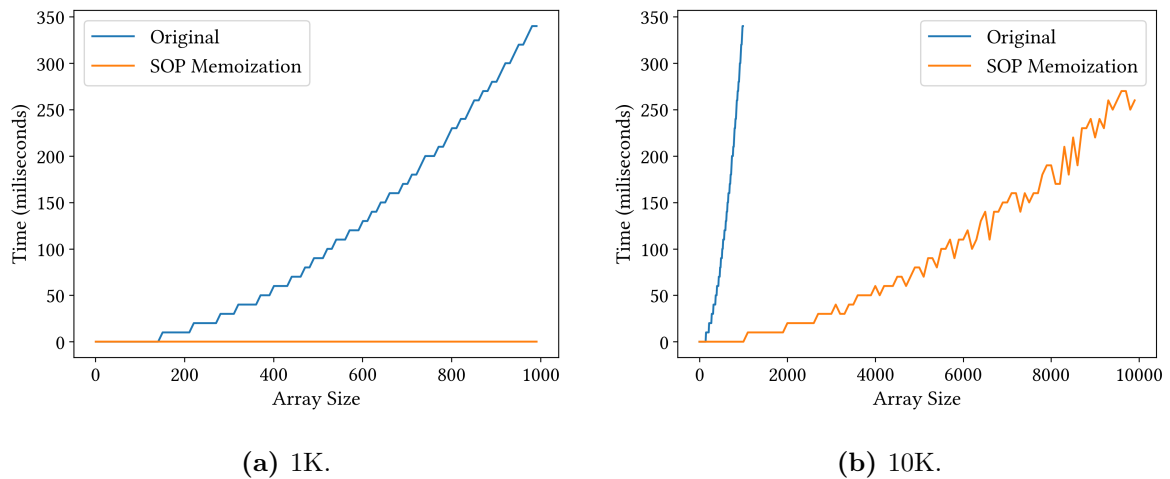


Figure 5.2: Running time of **Max Subarray Sum**.

gles, rectangles, etc), or the union, intersection or difference of other figures. Given a set of points and figures, it finds which figures contain which points.

Lines: a classic hash-based solution to the problem of finding three colinear points within a set of 3D points.

Polygons: a simple drawing application that represents polygons as the hull of sequences of points. Again, for a set of points and polygons, it finds which polygons contains which points.

Submarines: a solution to the first three problems of the “*Advent of Code 2021*”, which involve controlling a list of “submarines”. Each submarine can have its state altered by some displacement operation such as “forward” or “submerge”.

Treesort: a sorting algorithm that arranges objects in a tree. Trees are either leaf nodes, or internal nodes formed by an element and two subtrees.

The six benchmarks used in this Section create objects of varying complexity. Programs **Lines** and **Submarines** create very simple objects, whose attributes are all Primitive types. Programs **Polygons** and **Figures** create objects that point to other objects; yet, their structure is shallow: the depth of the object tree has height at most 2. Finally, programs **Eval** and **Treesort** create deep object-oriented structures, where an object can point to another, and these links might repeat an unbounded number of times. These benchmarks shall illustrate the shortcomings of memoization of mutable structures: the need to compute complex hash functions, for instance. The source code for the programs presented here is available in Appendix A.

Discussion Figure 5.3 shows the running time and collision rate of memoized objects for each benchmark. Memoization proves beneficial in three benchmarks: **Figures**, **Lines**, and **Submarines**. In **Submarines**, the high reuse rate of memoized objects pays off because the problem involves a limited number of distinct “Movements” for submarines. These movements, as memoized objects, are frequently reused. Submarines themselves are also memoized but are quickly removed from the memoization table due to the copy-on-write semantics explained in Section 3.4.

In **Figures** and **Lines**, memoization is advantageous because the object construction functions are relatively costly. These functions take points as parameters and use them to create complex objects like shapes and three-dimensional lines. Since objects are not modified during benchmark execution, they remain in the memoization table.

All benchmarks show a consistent trend: the cost of our safe memoization approach is not higher than the cost of unsafe memoization. The difference between **Sop** and **Unsafe** memoization is often statistically insignificant. In at least one benchmark, **Polygons**, **Sop** memoization results in faster code (with a confidence level of 0.95). We speculate that this speed increase might be due to better cache locality: mutated objects are accessed more often, reducing accesses to the memoization table.

5.3 RQ3: The Impact of Hashing

In our experience, the main challenge in memoizing objects was the impact of *deep hashing*. Comparing memoized objects involves traversing the graph of values reachable from those objects. In practice, this traversal means computing deep hashes for objects. In other words, the hash code of an object o is determined by o 's internal state and by the hashes of the objects that o contains. This recursive nature of the hashing operation can heavily tax the memoization system.

Methodology Using the **Add** class from **Eval**, the program in Figure 5.4a creates arithmetic expressions whose size grows exponentially with the value of the input **len**. At each recursive call, we create a new object and compute its hash in order to place it in the MT. To do that, we take into account the value of the entire chain of **Add** objects.

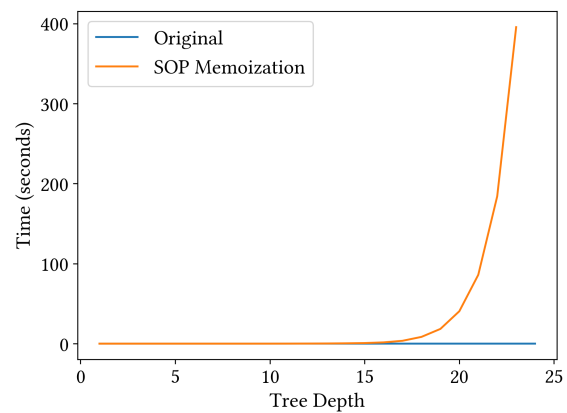
Discussion The time to hash an object is proportional to the size of that object. The size of an object, in turn, is the size of its primitive attributes, plus the size of the objects it contains. The chart in Figure 5.4b demonstrates that, in this example, memoization time grows exponentially with **len**—a direct consequence of the exponential growth of objects.

```

1  function gen(n)
2    if n == 1 then
3      Num(0)
4    else
5      let T = gen(n-1)
6      Add(T, T)
7    end
8  end
9  gen(N)

```

(a) Code added to **Eval**.



(b) Execution time by input size.

Figure 5.4: Example that illustrates the heavy impact of deep hashing on the memoization of complex objects.

5.3.1 Alternative Hashing Algorithms

The choice of a good hash function and an effective collision-handling strategy is crucial for maximizing the benefits of memoization in terms of both time and space complexity. Our implementation uses the SipHash 1-3 developed by Aumasson and Bernstein [4], the default one for hash tables in Rust. We shall compare this algorithm with four different approaches.

Discussion Figure 5.5 shows the running time of five different algorithms in two versions of the *eval* benchmark. In the case of (a) we exclusively use integers as keys. In (b), where we use compound objects as keys, SipHash 1-3 performed better than the alternatives. Although some of these algorithms are said to perform better in specific contexts—FxHash and FnvHash for primitive types, for example—our main use in Hush is hashing compound types. Therefore, we conclude that the default SipHash 1-3 performs better for our use case.

aHash: It tries to take advantage of the AES instruction set present in architectures such as X86_64 and ARM64.

FxHash: A simple hash function used internally in the Rust compiler; Fx meaning Firefox, where it was first seen.

SeaHash: It aims to balance speed and good distribution properties, making it suitable for performance-critical applications where hash functions are frequently called.

SipHash: Introduced by Aumasson and Bernstein [4] to provide a secure alternative to non-cryptographic hash functions for use in hash tables and other applications where collision resistance is important.

FnvHash: The FNV (Fowler-Noll-Vo) hash algorithm is a non-cryptographic hash function designed for fast hashing with good distribution properties. It is widely used for hash table implementations and other applications where fast hashing is required [10].

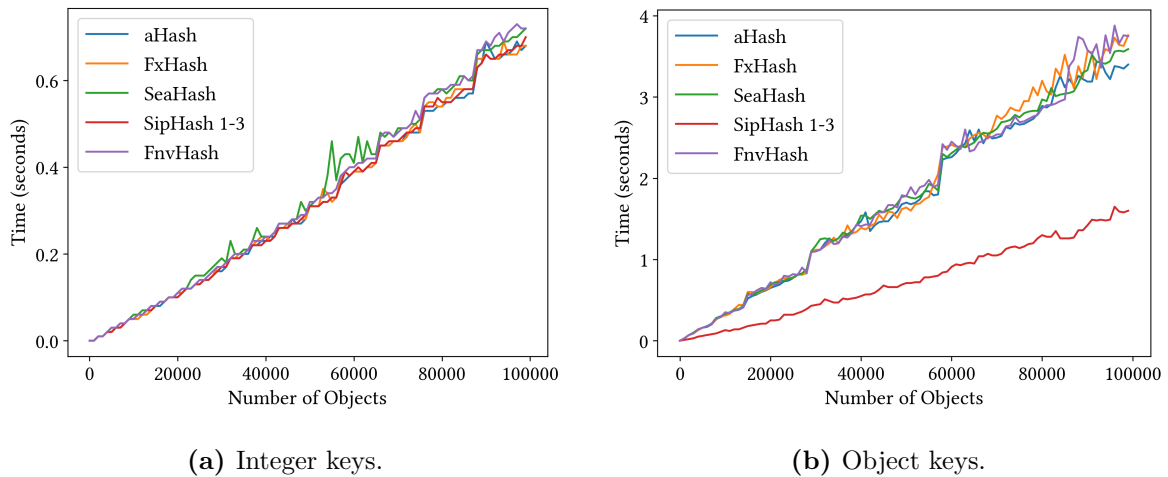


Figure 5.5: Execution time of **Lines** using different types of keys.

5.4 RQ4: The Impact of Tracing Memoized Data

Early in the integration of memoization in the Hush interpreter, we observed a dramatic slowdown in performance. This overhead was specially evident in the *levenshtein* benchmark. One key difference of this program is that it receives string inputs as opposed to integers such as in the other functional benchmarks (see Section 5.1). The research question in this section shall investigate the slowdown present in memoized functions with inputs there are scalar types.

Methodology As in Section 5.1, we run the benchmarks with inputs ranging from 0 to n . Each sample is run consecutively, so the instance n is run immediately after $n - 1$. This means the MT is filled with values obtained from the $n - 1$ instance before running n . Unlike previous experiments, we also run this benchmark with decreasing n . Meaning the $n - 1$ instance is run after n .

Two versions of the *levenshtein* benchmark are measured: (i) using the `memo` keyword, referred to as “builtin memoization”; (ii) implementing it without `memo`, in dynamic programming style, referred to as “handcrafted memoization”. Finally, we investigate the the slowdown by profiling the execution of *levenshtein* using the flamegraph¹ tool [21].

Discussion For small inputs, we start with just a few strings in the heap, allowing for relatively fast execution. As iterations progress, more strings are generated, increasing the workload for the garbage collector, which must reclaim unused memory. Since we are memoizing the results, all data remains in use. Consequently, the garbage collector scans a large collection of string objects without ever freeing them. As the number of strings grows, so does the time spent on garbage collection. Two main pieces of evidence support this hypothesis:

1. Execution time increases with the number of iterations, regardless of an increase or decrease in n . The only exception is when n is very small, as there are too few garbage collection rounds to affect execution time (see Figure 5.6).
2. The execution time spent by function is concentrated (up to 50%) on garbage collection routines, such as `drop` and `Trace` (see Figure 5.7).

¹Version 0.6.5

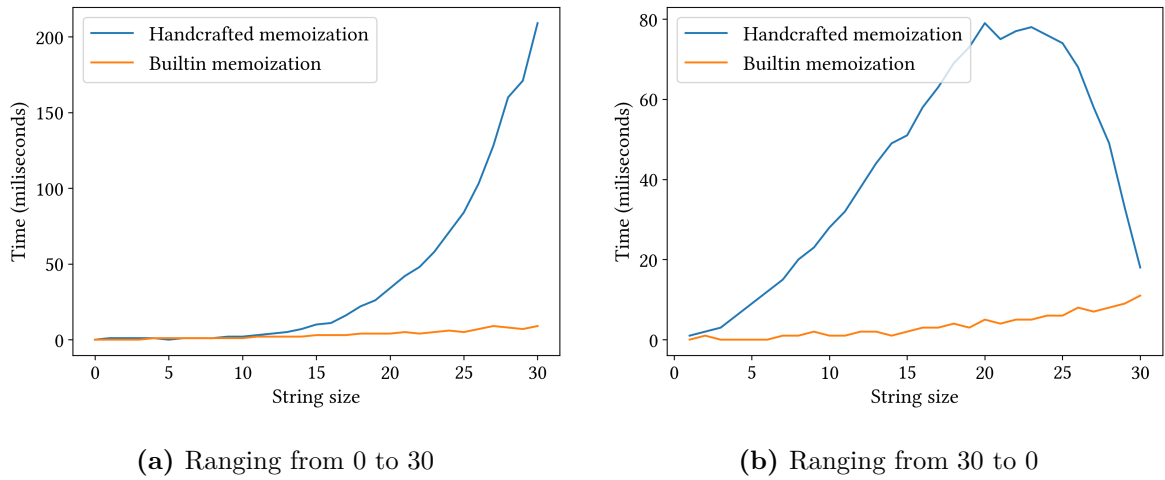


Figure 5.6: Execution time of the *levenshtein* benchmark, with inputs ranging from 0 to 30. Builtin memoization is using the `#[unsafe_ignore_trace]` macro from the Gc library.

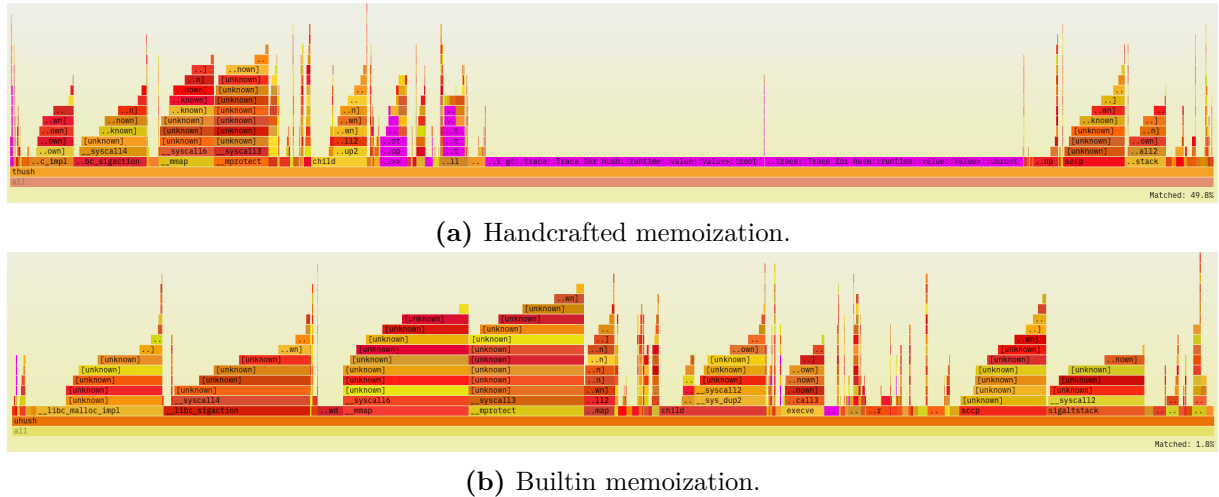


Figure 5.7: Flamegraph of functions executed in the Hush interpreter for the *levenshtein* benchmark, with purple indicating functions related to garbage collection. In (a), up to 50% of the total execution time is concentrated in garbage collection routines, while (b) spends only 1.8% of the total time in these functions.

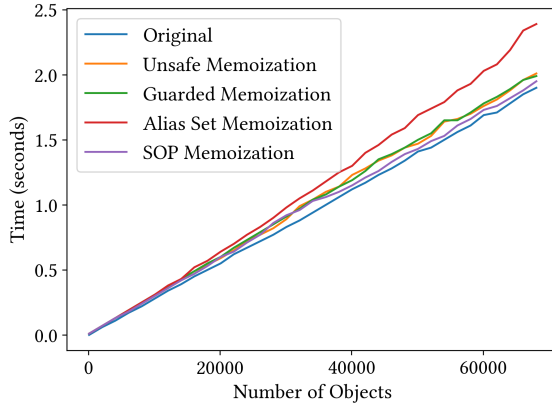
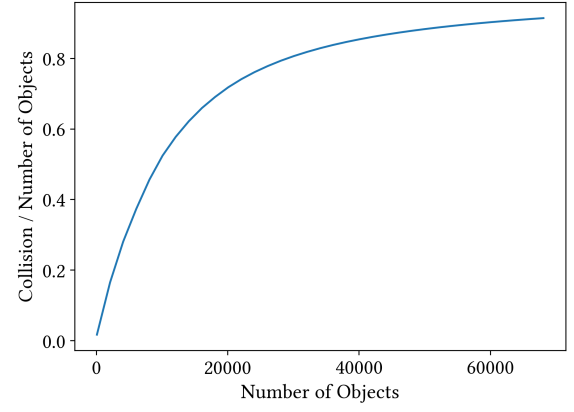
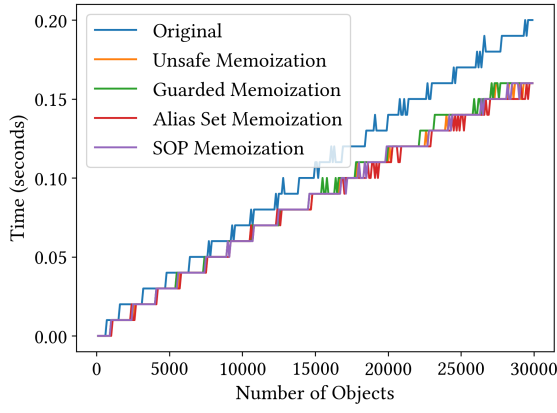
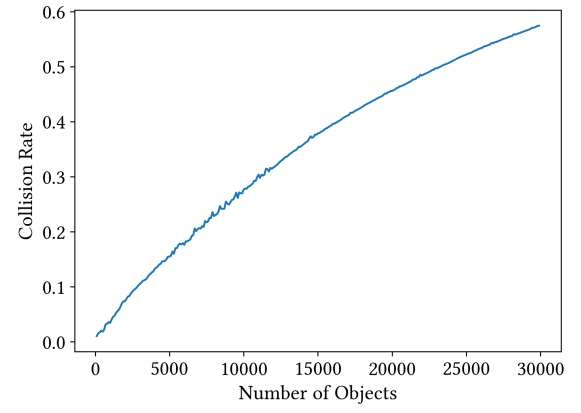
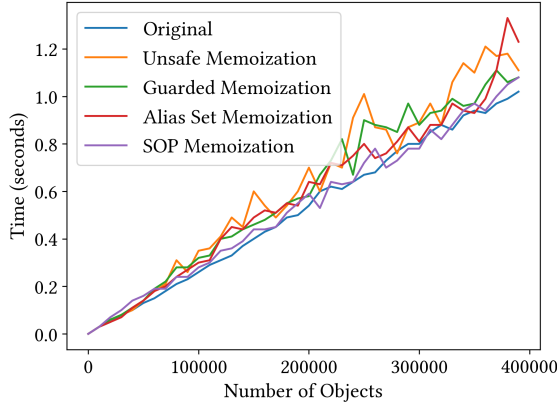
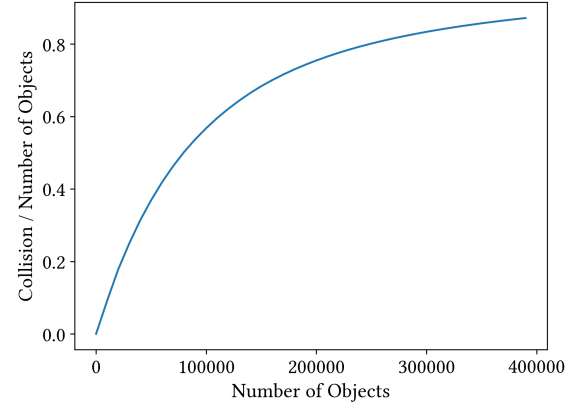
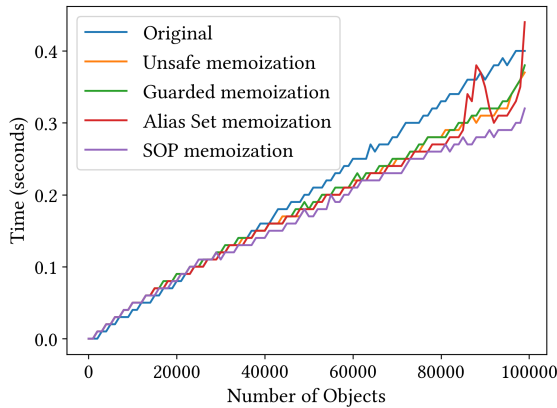
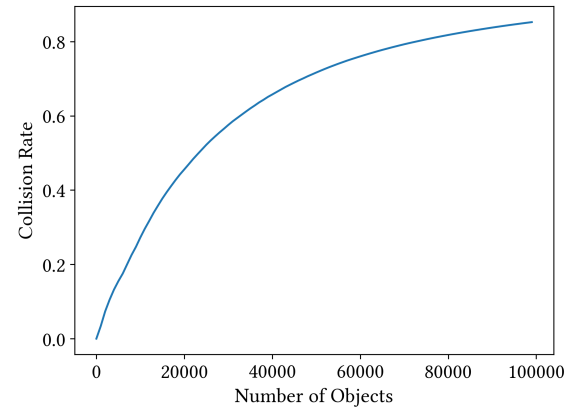
The key difference with builtin memoization is that it uses a special macro to indicate that memoized objects should not be traced by the GC. The macro `#[unsafe_ignore_trace]` is provided by the GC library used in the Hush interpreter. According to the second property of the specification in Chapter 4, memoized object must always be reachable. Therefore, tracing these objects unnecessarily increases the workload of the GC since they will never be collected. Illustrated in Figure 5.7, applying this change² to builtin memoization reduced the time spent on GC routines from 50% to 1.8% of the total execution time of this benchmark, compared to handcrafted memoization.

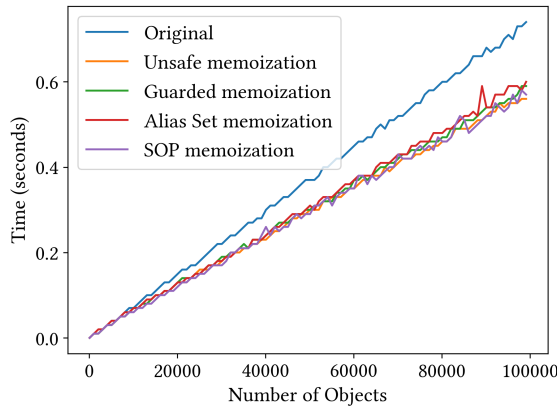
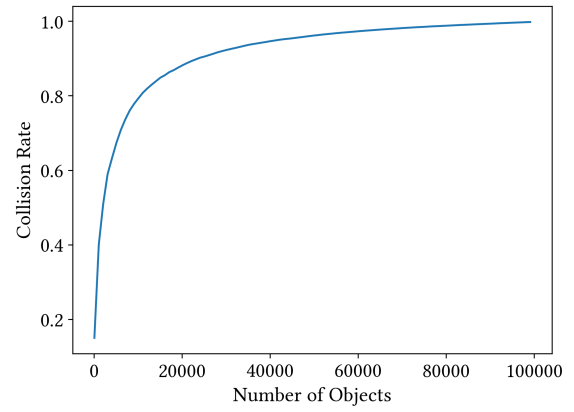
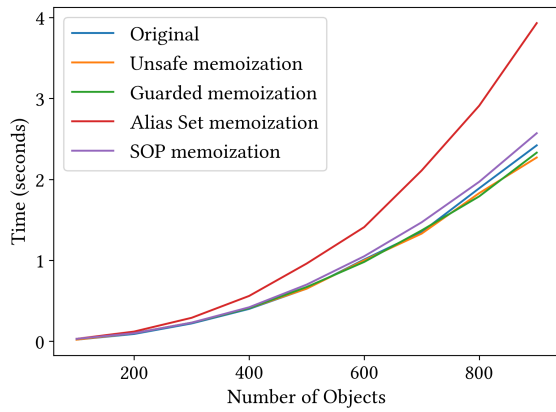
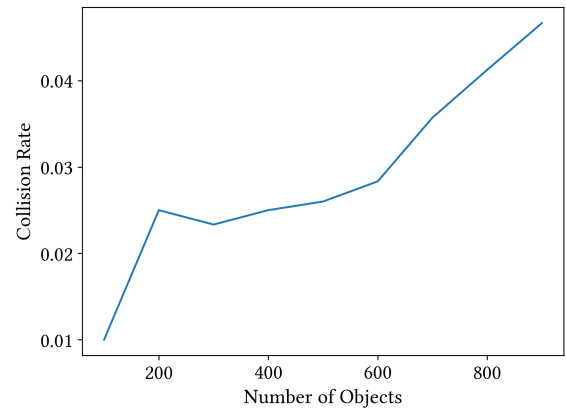
Memoizing methods in generally considered an anti-pattern in Python due to unclammed memory and increased GC overhead. This issue has been debated in CPython development forums³. For this reason, tuples and dictionaries containing only immutable

²Commit 624566c2b0427ea7b72d5aca7b99cd5122df13ab

³See discussion at <https://bugs.python.org/issue14775>

objects are not traced to reduce the cost of garbage collections. This is determined dynamically at every GC round by using the `untrack_dicts()` function present in the `gc` module of the CPython interpreter.

(a) Execution time of *polygons*.(b) Hit rate of *polygons*.(c) Execution time of *composite-figures*.(d) Hit rate of *composite-figures*.(e) Execution time of *eval*.(f) Hit rate of *eval*.(g) Execution time of *lines*.(h) Hit rate of *lines*.

(i) Execution time of *submarine*.(j) Hit rate of *submarine*.(k) Execution time of *treesort*.(l) Hit rate of *treesort*.**Figure 5.3:** Execution time and hit rate of object-oriented benchmarks.

Chapter 6

Conclusion

In this work we have presented a new technique to memoize functions that return mutable objects. Said technique consists in funneling references to the same object into shared pointers. In this way, we can adjust all the references to a given object with a single update operation, in case it is necessary to remove said object from the memoization table due to mutations. We have implemented our ideas in the Hush programming language; however, we believe that the general approach advocated in this work can be adapted to any other programming language that features mutable state.

6.1 Future Work

One potential enhancement to our memoization approach involves updating the memoization table upon object mutation. This could potentially increase object reuse if the memoization table contains more current data. For instance, in the scenario outlined in Section 3.1, including the modified object in the MT could enhance the versatility of our approach. However, assessing the impact of this modification on program performance and object reuse frequency is indispensable.

On Tracing Memoized Data Our implementation of memoization in Hush presents challenges concerning the GC, as discussed in Section 5.4. If not managed properly, the GC might inadvertently trace memoized data, potentially causing performance issues. This is a problem observed in other scripting languages which offer memoization constructs, such as Python. Consequently, we adjusted the tracing mechanism to instruct the GC to avoid memoized data stored in the MT. However, one possible concern of “switching off” the GC for certain objects the potential of *memory leaking*. Therefore, such implementation can still be up to debate.

The Cost of Hashing Throughout this work, we have memoized objects using deep hashes. The hash code of an object includes the hash of every subobject reachable through it. As seen in Section 5.3, this approach leads to significant slowdown when deeply nested objects are memoized. We believe it may be possible to improve this scenario by memoizing shallow copies of objects. This is a research avenue we hope will be explored in the future.

Automatic Memoization This work implements a new approach for memoization; however, it is up to the programmer to decide if memoization should be performed or not. There exists already previous research that tries to implement memoization automatically. We believe that our work expands the surface of possibilities for these lines of research. This happens because the automatic memoization of mutable objects presents

several challenges. These challenges include, for instance, the identification of complex constructors whose running time could be saved by memoization, the identification of deeply nested objects, which should not be memoized, and the identification of immutable objects, whose memoization tends to be inherently more advantageous.

6.2 Software

Our version of Hush is publicly available at <https://github.com/caioraposo/hush>. Although it still offers room for improvement, we believe that it can be used as a first proof of concept of the memoization of mutable objects.

Bibliography

- [1] Umut A. Acar, Amal J. Ahmed, and Matthias Blume. “Imperative self-adjusting computation”. In: *ACM-SIGACT Symposium on Principles of Programming Languages*. 2008.
- [2] Umut A Acar, Guy E Blelloch, and Robert Harper. “Selective memoization”. In: *ACM SIGPLAN Notices* 38.1 (2003), pp. 14–25.
- [3] Priya Arundhati, Sisir Kumar Jena, and Santosh Kumar Pani. “Approximate function memoization”. In: *Concurrency and Computation: Practice and Experience* 34 (2022).
- [4] Jean-Philippe Aumasson and Daniel J. Bernstein. *SipHash: a fast short-input PRF*. Cryptology ePrint Archive, Paper 2012/351. <https://eprint.iacr.org/2012/351>. 2012. URL: <https://eprint.iacr.org/2012/351>.
- [5] Gabriel Bastos. *The Hush Programming Manual*. <https://hush-shell.github.io/>. Accessed: 2024-04-25. 2021.
- [6] Daniel Bovet and Marco Cesati. *Understanding The Linux Kernel*. Oreilly & Associates Inc, 2005. ISBN: 0596005652.
- [7] Daniel Brown and William R Cook. *Monadic memoization mixins*. Citeseer, 2007.
- [8] Paul R. Calder and Mark A. Linton. “Glyphs: flyweight objects for user interfaces”. In: *UIST*. Snowbird, Utah, USA: Association for Computing Machinery, 1990, pp. 92–101. ISBN: 0897914104. DOI: 10.1145/97924.97935. URL: <https://doi.org/10.1145/97924.97935>.
- [9] Sanjoy Dasgupta, Christos Papadimitriou, and Umesh Vazirani. *Algorithms*. McGraw-Hill Higher Education, 2006, p. 171.
- [10] Donald E. Eastlake et al. “The FNV Non-Cryptographic Hash Algorithm”. In: 2024. URL: <https://datatracker.ietf.org/doc/html/draft-eastlake-fnv-22>.
- [11] Matthew A. Hammer et al. “Adapton: composable, demand-driven incremental computation”. In: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2014).
- [12] Robert Harper. “Self-adjusting computation”. In: *Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science, 2004*. (2004), pp. 254–255.
- [13] Daniel Jackson. “Alloy: a language and tool for exploring software designs”. In: *Commun. ACM* 62.9 (2019), pp. 66–76. ISSN: 0001-0782. DOI: 10.1145/3338843. URL: <https://doi.org/10.1145/3338843>.
- [14] Daniel Jackson. *Software Abstractions: logic, language, and analysis*. MIT press, 2012.
- [15] Kazutaka Kamimura et al. “A Speed-up Technique for an Auto-Memoization Processor by Reusing Partial Results of Instruction Regions”. In: *2012 Third International Conference on Networking and Computing* (2012), pp. 49–57.

- [16] James Kominick. *Cached*. <https://github.com/jaemk/cached>. Accessed: 2024-04-28. 2018.
- [17] Leslie Lamport. “The temporal logic of actions”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 16.3 (1994), pp. 872–923.
- [18] Zhenhong Liu et al. “AxMemo: Hardware-Compiler Co-Design for Approximate Code Memoization”. In: *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)* (2019), pp. 685–697.
- [19] Donald Michie. ““Memo” Functions and Machine Learning”. In: *Nature* 218 (1968), pp. 19–22.
- [20] Peter Norvig. “Techniques for Automatic Memoization with Applications to Context-Free Parsing”. In: *International Conference on Computational Logic*. 1991.
- [21] Dirkjan Ochtman. *[cargo-]flamegraph*. <https://github.com/flamegraph-rs/flamegraph>. Accessed: 2024-05-27. 2019.
- [22] Pedro Pinto and João MP Cardoso. “A methodology and framework for software memoization of functions”. In: *Proceedings of the 18th ACM International Conference on Computing Frontiers* (2021).
- [23] Hugo Rito and João Cachopo. “Memoization of methods using software transactional memory to track internal state dependencies”. In: *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java*. 2010, pp. 89–98.
- [24] Harald Søndergaard and Peter Sestoft. “Referential transparency, definiteness and unfoldability”. In: *Acta Informatica* 27 (1990), pp. 505–517.
- [25] Mirko Stoffers et al. “Automated memoization for parameter studies implemented in impure languages”. In: *Proceedings of the 2016 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*. 2016, pp. 221–232.
- [26] Mirko Stoffers et al. “On automated memoization in the field of simulation parameter studies”. In: *ACM Transactions on Modeling and Computer Simulation (TOMACS)* 28.4 (2018), pp. 1–25.
- [27] Yican Sun, Xuanyu Peng, and Yingfei Xiong. “Synthesizing Efficient Memoization Algorithms”. In: *Proceedings of the ACM on Programming Languages* 7 (2023), pp. 89–115.
- [28] Gurram Sunitha et al. “Dynamic Programming With Python”. In: *Advanced Applications of Python Data Structures and Algorithms*. IGI Global, 2023, pp. 102–122.
- [29] Arjun Suresh, Erven Rohou, and André Seznec. “Compile-time function memoization”. In: *Proceedings of the 26th International Conference on Compiler Construction* (2017).
- [30] Arjun Suresh et al. “Intercepting Functions for Memoization”. In: *ACM Transactions on Architecture and Code Optimization (TACO)* 12 (2015), 18:1–18:23.
- [31] David Svoboda and Lutz Wrage. “Pointer ownership model”. In: *HICSS*. IEEE. 2014, pp. 5090–5099.
- [32] Georgios Tziantzoulis, Nikolaos Hardavellas, and Simone Campanoni. “Temporal Approximate Function Memoization”. In: *IEEE Micro* 38 (2018), pp. 60–70.

-
- [33] Guowei Zhang and Daniel Sánchez. “Leveraging Hardware Caches for Memoization”. In: *IEEE Computer Architecture Letters* 17 (2018), pp. 59–63.

Appendix A

Object-Oriented Benchmarks

This appendix introduces six object-oriented programs that will be used for benchmark evaluation. Each program is designed to test different aspects of performance and resource utilization, particularly in scenarios involving memoization, dynamic memory management, and object interactions. The programs are structured using object-oriented principles, including encapsulation, inheritance, and polymorphism, to model complex data structures and operations.

```

1  # The rectangle is represented by bottom left point, width, and height.
2  let Rectangle = memo function(x, y, w, h)
3    @[
4      _x: x, _y: y, _w: w, _h: h,
5
6      contains: function(x, y)
7        if x < self._x or x > self._x + self._w or
8          y < self._y or y > self._y + self._h then false
9        else true
10       end
11     end,
12   ]
13 end
14
15 let Circle = memo function(x, y, r)
16   @[
17     _x: x, _y: y, _r: r,
18
19     contains: function(x, y)
20       if pow(self._r, 2) < pow(self._x - x, 2) + pow(self._y - y, 2) then false
21       else true
22       end
23     end,
24   ]
25 end
26
27 function union(a, b)
28   if self._a.contains(x, y) or self._b.contains(x, y) then
29     true
30   else
31     false
32   end
33 end
34
35 function difference(a, b)

```

```

36   if self._a.contains(x, y) and not self._b.contains(x, y) then
37       true
38   else
39       false
40   end
41 end

```

Listing A.1: Figures: an application that represents figures as either primitive shapes (circles, triangles, rectangles, etc), or the union, intersection or difference of other figures. Given a set of points and figures, it finds which figures contain which points.

```

1  let Line = memo function(a, b, x, y)
2    let line = @[]
3
4    # handle division by zero
5    if a - x != 0 then
6        line["a"] = (b - y) / (a - x)
7    else
8        line["a"] = (b - y)
9    end
10   line["b"] = b - line["a"] * a
11
12   line
13 end

```

Listing A.2: Lines: a classic hash-based solution to the problem of finding three colinear points within a set of 3D points.

```

1  let Point = memo function(x, y, z)
2    @[
3      _x: x,
4      _y: y,
5      _z: z,
6    ]
7  end
8
9  function Polygon(n)
10   let points = []
11   for _ in std.range(0, n, 1) do
12       let x = std.randint(0, X)
13       let y = std.randint(0, Y)
14       let z = std.randint(0, Z)
15       let p = Point(x, y, z)
16       std.push(points, p)
17   end
18   points
19 end

```

Listing A.3: Polygons: a simple drawing application that represents polygons as the hull of sequences of points. For a set of points and polygons, it finds which polygons contains which points.

```

1  let Submarine = memo function(x, y)
2    @[

```

```

3     _x: x,
4     _y: y,
5   ]
6 end
7
8 # Note that we don't the base class
9 let Move = function(sub)
10   @[
11     _sub: sub,
12
13     move: function()
14       # Should be implemented in the subclasses.
15       std.panic("method not implemented")
16     end,
17   ]
18 end
19
20 let Forward = memo function(sub)
21   let super = Move(sub)
22
23   super.move = function(n)
24     Submarine(self._sub._x + n, self._sub._y)
25   end
26
27   super
28 end
29
30 let Up = memo function(sub)
31   let super = Move(sub)
32
33   super.move = function(n)
34     Submarine(self._sub._x, self._sub._y - n)
35   end
36
37   super
38 end
39
40 let Submerge = memo function(sub)
41   let super = Move(sub)
42
43   super.move = function(n)
44     Submarine(self._sub._x, self._sub._y + n)
45   end
46
47   super
48 end

```

Listing A.4: Submarines: a solution to the second day problem of the “*Advent of Code 2021*”, which involve controlling a list of “submarines”. Each submarine can have its state altered by some displacement operation such as “forward” or “submerge”.

```

1 let Rational = memo function(p, q)

```

```
2  @[
3    _p: std.float(p),
4    _q: std.float(q),
5    value: function()
6      self._p / self._q
7    end
8  ]
9  end
10
11 let Node = function(n)
12   @[
13     _n: n,
14     _left: nil,
15     _right: nil,
16   ]
17 end
18
19 let BinarySearchTree = function()
20   @[
21     _root: nil,
22
23     _push_r: function(r, n)
24       if r == nil then
25         r = Node(n)
26       elseif r._n.value() >= n.value() then
27         r._left = self._push_r(r._left, n)
28       else
29         r._right = self._push_r(r._right, n)
30       end
31       r
32     end,
33
34     push: function(n)
35       self._root = self._push_r(self._root, n)
36     end,
37
38     inorder: function()
39       let stack = []
40       let sorted = []
41       let node = self._root
42       while not std.is_empty(stack) or node != nil do
43         if node != nil then
44           std.push(stack, node)
45           node = node._left
46         else
47           node = std.pop(stack)
48           std.push(sorted, node._n.value())
49           node = node._right
50         end
51       end
52       sorted
```

```

53     end,
54   ]
55 end

```

Listing A.5: Treesort: a sorting algorithm that arranges objects in a tree. Trees are either leaf nodes, or internal nodes formed by an element and two subtrees.

```

1  let Num = memo function(n)
2    @[
3      _n: n,
4
5      eval: function()
6        self._n
7      end
8    ]
9  end
10
11 let Add = memo function(a, b)
12   @[
13     _a: a,
14     _b: b,
15
16     eval: function()
17       self._a.eval() + self._b.eval()
18     end
19   ]
20 end
21
22 let Sub = memo function(a, b)
23   @[
24     _a: a,
25     _b: b,
26
27     eval: function()
28       self._a.eval() - self._b.eval()
29     end
30   ]
31 end
32
33 let Mul = memo function(a, b)
34   @[
35     _a: a,
36     _b: b,
37
38     eval: function()
39       self._a.eval() * self._b.eval()
40     end
41   ]
42 end
43
44
45 let Div = memo function(a, b)

```

```
46  @[
47    _a: a,
48    _b: b,
49
50    eval: function()
51      if self._b.eval() == 0 then
52        self._a.eval()
53      else
54        self._a.eval() / self._b.eval()
55      end
56    end
57  ]
58 end
```

Listing A.6: Eval: an evaluator of arithmetic expressions. Expressions are either numbers of variables, or the composition of other expressions, such as addition, multiplication or division of subexpressions.

Appendix B

Memoization Model in Alloy

This Appendix presents a model of the memoization of mutable objects on a programming language. This model should specify our general technique for memoizing objects and attest its soundness. The Hush interpreter is where the actual implementation of this technique resides. Therefore, our model in Alloy shall have the necessary components for representing the semantics of the concrete implementation.

To overcome the challenge of mutability, we propose an approach to memoization which is based on Copy-on-Write. As such, whenever an object is cached in the memoization table, it will never be modified. Instead, we create a copy, modify the copy, and change the reference to the previous object to this new copy. Our specification in Alloy will show why this approach is valid.

Firstly, we have the following signatures:

Reference: This represents a variable identifier, such as `p0` and `p1` in the Python example.

Object: An object allocated in memory (what `p0` and `p1` are pointing to).

ObjectStatus: Can either be reachable or unreachable. Unreachable meaning there is no **Reference** pointing to it, and neither a reachable object.

Params: The arguments passed down to the constructor of an object.

MemoTable: A global memoization table which maps **Params** to **Object**.

Operator: The set of possible operations in the system.

Trace: Which operation was most recently executed.

Most operations are performed in references. Which in turn may or may not change objects in memory.

`read`: Read the object a reference is pointing to (`p0.get()`); nothing changes.

`write`: Modify the object (`p0.set()`). In Alloy, we model it by changing what the object points to.

`copy_on_write`: A variant of `write` which is safe for memoization.

`drop`: A reference going out of scope. If nothing else is pointing to that object, then it becomes unreachable.

`alias`: Create an alias such as: `a = b`.

We believe that these predicates are enough to create the scenarios of interest. We propose six properties which must be satisfied by our model, aiming at showing that our implementation of memoization is sound. Some properties, such as Property 1, helped to “debug” the model. Property 2, ensures the garbage-collector won’t deallocate things that are in the memoization table, and Property 6 ensures it will deallocate other objects. The other properties verify that memoization is working as intended.

```

1  abstract sig ObjectStatus {}
2  one sig Reachable, Unreachable extends ObjectStatus {}
3
4  var sig Object {
5    var status: one ObjectStatus,
6    var points: set Object
7  }
8
9  var sig Params {
10   var memo: one Object
11 }
12
13 var sig Reference {
14   var alias: set Reference,
15   var points: lone Object
16 }
17
18 abstract sig Operator {}
19 one sig WRITE, READ, NEW, ALIAS, DROP, COLLECT, DECLARE extends Operator {}
20
21 one sig Track {
22   var op: lone Operator,
23 }
24
25 pred nothingChanges[] {
26   Object' = Object
27   Object.points' = Object.points
28   status' = status
29   Params' = Params
30   memo' = memo
31   Reference' = Reference
32   all r: Reference | r.points' = r.points
33   alias' = alias
34 }
35
36 /* Operators */
37
38 /* Change the object pointed by `s`. This is unsafe for memoization. */
39 pred write [r: Reference, o: Object] {
40   // Pre conditions
41   r.points = o
42
43   // Post conditions
44   o.points' != o.points // change what `o` is pointing to.
45   o.status' = o.status
46   r.points' = o
47   r.alias' = r.alias
48   Track.op' = WRITE
49
50   // Frame conditions
51   Object' = Object
52   Reference' = Reference
53   Params' = Params
54   all o: Object - o | o.status' = o.status and o.points' = o.points
55   all r: Reference - r | r.alias' = r.alias and r.points' = r.points
56   all p: Params | p.memo' = p.memo
57 }
58
59 pred copy_on_write [r: Reference, o: Object] {
60   // Pre conditions
61   r.points.status = Reachable

```

```

62
63 // Post conditions
64 copy[r.points, o] // o is a copy of r.points
65 r.points' = o
66 o.points' != o.points // change the copy.
67 r.alias' = r.alias
68 Track.op' = WRITE
69
70 // Frame conditions
71 all o: Object - o | o.status' = o.status and o.points' = o.points
72 all r: Reference - r | r.alias' = r.alias and r.points' = r.points
73 all p: Params | p.memo' = p.memo
74 }
75
76 pred read [r: Reference] {
77   Track.op' = READ
78
79   // Frame conditions
80   nothingChanges[]
81 }
82
83 pred new[r: Reference, p: Params, o: Object] {
84   // Pre conditions
85   r !in Reference
86
87   // Post conditions
88   Reference' = Reference + r
89   Object' = Object + o
90   Params' = Params + p
91   p in Params => {
92     o in Object
93   } else {
94     o !in Object
95   }
96   o.status' = Reachable
97   r.points' = o
98   p.memo' = o
99   r.alias' = r
100   Track.op' = NEW
101
102   // Frame conditions
103   all r: Reference | r.alias' = r.alias and r.points' = r.points
104   all o: Object | o.status' = o.status and o.points' = o.points
105   all p: Params - p | p.memo' = p.memo
106 }
107
108 /* Create the alias: a = b. */
109 pred alias [a: Reference, b: Reference] {
110   // Pre conditions
111   some b.points
112
113   // Post conditions
114   a.alias' = a.alias + b
115   b.alias' = b.alias + a
116   a.points' = b.points
117   Track.op' = ALIAS
118
119   // Frame conditions
120   b.points' = b.points
121   all o: Object | o.status' = o.status and o.points' = o.points
122   all r: Reference - a - b | r.alias' = r.alias and r.points' = r.points
123   all p: Params | p.memo' = p.memo
124 }
125
126 /* Represents a reference going out of scope. */
127 pred drop [r: Reference, o: Object] {
128   // Pre conditions
129   r.points = o
130
131   // Post conditions

```

```

132   r !in Reference'
133
134   no t: Reference - r | t.points = o
135   no p: Object | p.status = Reachable and o in p.*points implies {
136     o.status' = Unreachable
137   } else {
138     o.status' = Reachable // o stays reachable.
139   }
140   Track.op' = DROP
141   o.status' = Unreachable
142
143   // Frame conditions
144   Object' = Object
145   Params' = Params
146   o.points' = o.points
147   all o: Object - o | o.status' = o.status and o.points' = o.points
148
149   all r: Reference - r | r.alias' = r.alias and r.points' = r.points
150   all p: Params | p.memo' = p.memo
151 }
152
153 pred collect [] {
154   // Pre conditions; all objects that o points to must be unreachable as well.
155   some o: Object | o.status = Unreachable
156
157   // Post conditions
158   all o: Object | o.status = Unreachable => o !in Object'
159   Track.op' = COLLECT
160
161   // Frame conditions
162   all o: Object | o.status != Unreachable => o.status' = o.status
163   all r: Reference | r.alias' = r.alias and r.points' = r.points
164 }
165
166 /* States that o1 is a copy of o0. */
167 pred copy [o0: Object, o1: Object] {
168   // Pre conditions
169   o1 !in Object
170
171   // Post conditions
172   o1 in Object'
173   o1.status' = o0.status
174   o1.points' = o0.points
175
176   // Frame conditions
177   all o: Object - o1 | o.status' = o.status and o.points' = o.points
178   // alias and memo are not here because we shall change them in the write predicate.
179 }
180
181 /* Properties */
182
183 /* References can only point to objects that are reachable. */
184 pred p1 [] {
185   all r: Reference | one r.points => r.points.status = Reachable
186 }
187
188 /* Objects stored in the memoization table are always reachable.
189    For any binary relation r, the domain and range of r
190    is r.univ and univ.r, respectively.
191 */
192 pred p2 [] {
193   all o: univ.(memo) | always o.status = Reachable
194 }
195
196 /* Two objects created via a memoized function with the same parameters are the same, until one
197    ↪ of them is modified. */
198 pred p3 [] {
199   some disj s0, s1: Reference | some p: Params | some disj o0, o1: Object | {
200     new[s0, p, o0] and (not write[s0, o0] until new[s1, p, o1]) => o0 = o1
201   }
202 }

```

```

201 }
202
203 /* Two objects created via a memoized function with the same parameters remain the same if one
    ↪ of them suffers an unsafe write. */
204 pred p4 [] {
205   some s0, s1: Reference | some p: Params | some o0, o1: Object | {
206     new[s0, p, o0] and write[s0, o0] and new[s1, p, o1] => o0 = o1
207   }
208 }
209
210 /* Two objects created via a memoized function with the same parameters are no the same after
    ↪ one of them suffers a copy-on-write. */
211 pred p5 [] {
212   some s0, s1: Reference | some p: Params | some o0, o1: Object | {
213     new[s0, p, o0] and copy_on_write[s0, o0] and new[s1, p, o1] => o0' != o1'
214   }
215 }
216
217 /* Every unreachable object will be evetually collected. */
218 pred p6 [] {
219   all o: Object | o.status = Unreachable => eventually collect[] and no o.status'
220 }
221
222 pred p7 [] {
223   all disj p, q: Params | some p.memo implies p.memo !in q.memo
224 }
225
226 /* Initial state conditions */
227
228 pred init[] {
229   no Object
230   no Reference
231   no Params
232   no Track.op
233 }
234
235 /* Transition relation */
236
237 pred trans [] {
238   (some r: Reference' | some p: Params' | some o: Object' | new [r, p, o])
239   or
240   (some r: Reference | read [r])
241   or
242   (some r: Reference | some o: Object | drop [r, o])
243   or
244   (some r: Reference | some o: Object | write [r, o])
245   or
246   (collect[])
247 }
248
249 /* System predicate */
250
251 pred System {
252   init
253   always trans
254   eventually (some r: Reference' | some p: Params' | some o: Object' | new [r, p, o])
255 }
256
257 run execution { System } for 8
258
259 /* Assertions */
260 assert a1 { System => always p1 }
261 assert a2 { System => always p2 }
262 assert a3 { System => always p3 }
263 assert a4 { System => always p4 }
264 assert a5 { System => always p5 }
265 assert a6 { System => always p6 }
266 assert a7 { System => always p7 }
267 check a1
268 check a2

```

269 [check](#) a3
270 [check](#) a4
271 [check](#) a5
272 [check](#) a6
273 [check](#) a7