Memoization of Mutable Objects

Caio Raposo
UFMG
Brazil
caioraposo94@gmail.com

Fernando Magno Quintão Pereira
Federal University of Minas Gerais
Brazil
fernando@dcc.ufmg.br

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 paper 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.

CCS Concepts: • **Software and its engineering → Compilers**.

Keywords: Hash, Synthesis, Specialization

1 Introduction

Memoization is a technique designed to optimize the execution of functions by caching their results [1, 9, 13]. This optimization stores the results of function calls and returns the cached result when the same inputs occur again, instead of recomputing the result. This technique is available in many different programming languages. As an example, Python provides memoization via annotations such as `@functools.cache` or `@functools.lru_cache` [20], and Haskell achieves it through the `memoFix` function wrapper from the `Data.Function.Memoize` library [5].

Mutability and Referential Transparency. Memoization is typically applied onto functions that are *referentially transparent* and that return *immutable values*. A function f is termed referentially transparent if two different invocations of f with the same input always yield the same output [17]. Referential transparency is a required property for memoization because only the first activation of a memoized function will happen during the execution of a program – further invocations will be replaced with cached values.

The need for immutability appears because attempts to modify a memoized value might impact every reference to that cached value. As a consequence of this limitation, memoization is seldom used in object-oriented programs. Objects typically encode mutable state; hence, attempts to mutate a memoized object might also mutate other instances of that object created with the same parameters. As an illustration, Section 2.3 provides an example of how memoization of object constructors compromise the correctness of Python programs. This paper describes a technique that relaxes this second requirement—immutability; hence, allowing the memoization of routines that create objects.

The Contributions of this Work. This paper describes a methodology to implement memoization of routines that return mutable values. This methodology consists of the combination of *shared-ownership pointers* [21] and the *copy-on-write* [4] policy. We postpone implementation details to Section 3; however, in a nutshell, our implementation of memoization brings the guarantees that we enumerate below, where o is an object, and r is a reference that points to o:

1. Every object o is memoized upon creation.
2. If a pointer r is used to give access to an operation that mutates o, then o is no longer memoized.
3. If a pointer r is not used to give access to an operation that mutates o, then o remains memoized.
4. If o is memoized, then any property in o can be accessed through r with two pointer dereferences.
5. If o is not memoized, then accesses to o via r happens without performance penalty, with one dereference.

Section 4 formalizes these guarantees on the Alloy [11] specification language.

Summary of Results. We have implemented the proposed methodology in the Hush programming language [2]. Hush is a scripting language, with syntax similar to Lua's, which Section 2.1 briefly reviews. We chose to work on Hush because it provides support to object-oriented programming by creating objects as closures. Thus, by memoizing routines that return such closures, we are effectively memoizing object constructors. Experiments discussed in Section 5 show that our implementation of memoization is only marginally slower than the use of unsafe memoization (as implemented

in Python, for instance, and explained in Section 2.3). Furthermore, depending on the problem at hand, safe memoization brings all the benefits of traditional memoization—of immutable values—onto an object-oriented setting where objects encode mutable state. More importantly, the addition of safe memoization to routines that create objects brings in a beautiful consequence: in practice, it makes the *flyweight* design pattern [6] native in these languages.

2 Background

This section has three goals. First, Section 2.1 quickly describes the Hush programming language. Familiarity with Hush is not required to understand the concepts presented in this paper; therefore, we only introduce the minimal syntax relevant to object creation. Section 2.2 explains how memoization works. Although memoization is a well-known and popular technique, we explain it to ensure that this material is self-contained. Finally, Section 2.3 discusses why memoizing mutable values is challenging, using, to this end, examples written in the Python programming language.

2.1 The Hush Programming Language

Hush is a shell scripting language whose syntax is based on Lua. As a shell language, it provides constructs for invoking and interconnecting external programs. The language's runtime environment is implemented in Rust. In terms of language design, Hush provides static scoping, strong dynamic typing, garbage collection and first class functions. Example 2.1 shows how Hush implements objects.

Example 2.1. Hush does not provide users with builtin syntax to create objects; rather, objects are modeled as dictionaries: keys are field names, and values are attributes or methods. Figure 1 shows a function that emulates a class. It takes the argument n, and uses it to construct a new object.

```
01 let Counter = function(n)
02   @[
03     _start: n,
04
05     get: function()
06       self._start
07     end,
08
09     inc: function()
10       self._start = self._start + 1
11     end,
12   ]
13 end
```

This is the syntax of a dictionary. Function `Counter` returns an object as a dictionary.

The functions bound to get and inc are closures. They contain one free variable, _start, which refers to "n", the argument originally passed to the function.

Figure 1. Function that fills the role of a class in Hush.

The approach seen in Example 2.1 to create objects is not original: there are other programming languages that create objects as closures. In particular, the implementation of classes in Simula 67 followed a similar design: a class was essentially a function that constructs and returns objects, as Dahl and Nygaard [7] explain in Section 2.7 of their work.

2.2 Memoization

Memoization, a term potentially coined by Michie [15], is an optimization technique that consists in caching the results of referentially transparent function calls. By avoiding recomputations, memoization is able to reduce the asymptotic complexity of some algorithms, as Example 2.2 shows.

Example 2.2. Figure 2 shows implementations, in Hush and Python, of an algorithm that solves the Knapsack Problem, which is stated as follows: "*Given a set of items, each with a weight (*`w`*) and a value (*`v`*), determine which items to include in the collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.*" [10]. The Knapsack Problem is typically solved via dynamic programming: solutions of smaller instances of the problem are used towards solving larger instances. To this end, a table is used to store intermediate solutions. Memoization gives this table to the programmer for free. In Hush, memoization is achieved via the `memo` qualifier (seen in Line 01 of the Hush implementation). Python provides multiple alternatives to memoize functions. In Figure 2, we use the `@cache` annotation, available in the `functools` library.

Hush

```
01 memo function ks(c, n)
02   if c == 0 or n == 0 then
03     0
04   elseif w[n-1] > c then
05     ks(c, n-1)
06   else
07     let r = ks(c-w[n-1], n-1)
08     max(
09       v[n-1] + r,
10       ks(c, n-1)
11     )
12   end
13 end
```

Python

```
01 @cache
02 def ks(c, n):
03   if c == 0 or n == 0:
04     return 0
05   elif w[n-1] > c:
06     ks(c, n-1)
07   else:
08     r = ks(c-w[n-1], n-1)
09     max(
10       v[n-1] + r,
11       ks(c, n-1)
12     )
```

Figure 2. A memoized solution to the Knapsack Problem, implemented in Hush and in Python.

Memoization improves performance: caching the results of function calls avoids the need to recompute them. Figure 3 highlights this benefit. That chart compares the time t taken to solve the Knapsack Problem with the number n of different objects that can be packed together. We have that $t = O(2^n)$ in general; however, for memoized functions several of the calls to the `ks` routine, be it in Hush, be it in Python, can be performed in constant time. The SOP (Shared-Ownership

Pointer) memoization mentioned in Figure 3 is the approach that Section 3 explains. Figure 3 shows that Python tends to outperform Hush when memoization is not used. However, our implementation of memoization, at least when applied onto the routines in Figure 2, yields faster programs. This last observation is a consequence of engineering: whereas memoization in Python is implemented in the language itself as a library, our SOP memoization is implemented in Rust, as part of Hush runtime system.

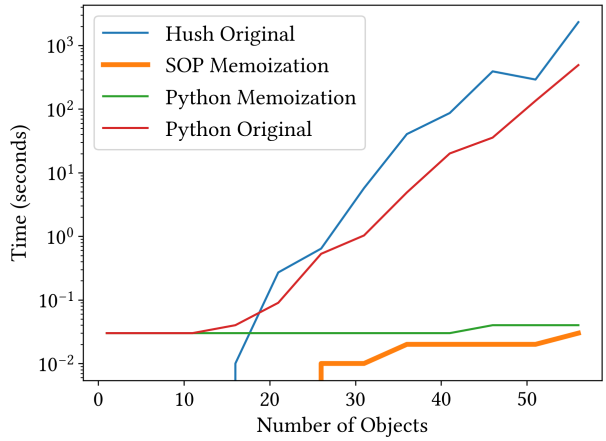


Figure 3. Comparison between memoized and non-memoized versions of the functions seen in Figure 2.

2.3 The Challenge of Mutability

The memoization of mutable values might lead to the design of incorrect programs, because it creates unwanted *aliases*. Aliasing is a feature of imperative programming languages that allows multiple pointers refer to the same memory location. In this context, Example 2.3 shows why the memoization of mutable objects might lead to the implementation of incorrect programs.

Example 2.3. Figure 4 shows a Python program that implements bidimensional points. Points are instances of the class `Point`. The annotation `@cache` forces the memoization of the `__init__` routine that creates objects. Thus, objects created with the same initializing parameters will be memoized. Consequently, references `p0` and `p1` point to the same object allocated in the memoization table. Once a property of `p1` is modified, the same property of `p0` is also affected. However, this behavior is erroneous, because the program does not contain any explicit copy operation (such as the statement `p0 = p1`) that causes these objects to alias.

3 Shared Ownership of Memoized Values

This section explains how we implement memoization of mutable values. Throughout this explanation, the reader should keep in mind that our approach asks for modifications in the language runtime system—it cannot be implemented as a library. This last observation comes from the fact that we change how properties of memoized objects are accessed: in our approach, accesses require one extra pointer dereference, in addition to the normal search for the property.

```
01 from functools import cache
02
03 @cache
04 class Point:
05   def __init__(self, x, y):
06     self.x = x
07     self.y = y
08 # Memoize:
09 p0 = Point(1, 1)
10 p1 = Point(1, 1)
11 # Mutate
12 p1.x = 2
13 # Pass...
14 assert(p1.x == 2)
15 # Fail!
16 assert(p0.x == 1)
```

Figure 4. Example where the memoization of a routine that creates mutable values causes the program to be incorrect.

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 `Arc<T>` (Atomic Reference Counted), which is the thread-safe version of `Rc<T>`.

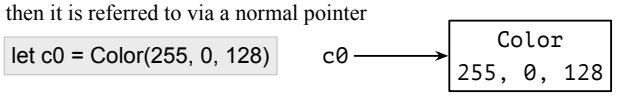
In this paper, shared-ownership pointers are not a resource to avoid memory leaks. Rather, they are a means to implement the "copy-on-write" policy of memoized objects. We have modified the Hush runtime environment to only refer to memoized objects through shared pointers. However, non-memoized objects are still accessed through plain references. Figure 5 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 5 (a):

- As seen in Figure 5 (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 5 (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 5 (d), if `p0` = `sop` is a reference to an object o, and o is updated, then o is copied into a

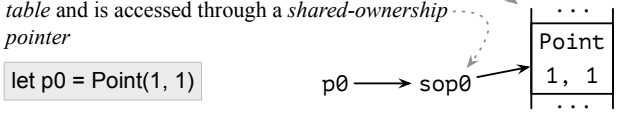
new object o', which is outside the memoization table. The shared reference sop is updated to point to o'.

(a)
```
01 let Color = function(red, green, blue)
02   @[r: red, g: green, b: blue,]
03 end
04
05 let Point = memo function(dim_x, dim_y)
06   @[x: dim_x, y: dim_y,]
07 end
```

(b) If an object is created without memoization, then it is referred to via a normal pointer



(c) Memoized object is stored in the *memoization table* and is accessed through a *shared-ownership pointer*



(d) If a memoized object is modified, then a copy of it is created *outside the memo table*, and the shared pointer is updated to refer to the copy

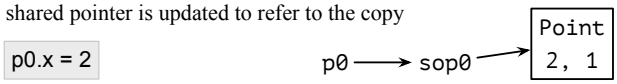


Figure 5. Distinction between non-memoized and memoized objects. The latter are accessed through a shared pointer, and stored in the memoization table unless modified.

If two pointers refer to the same memory location, then we say that they are *aliases*. In this paper, 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.

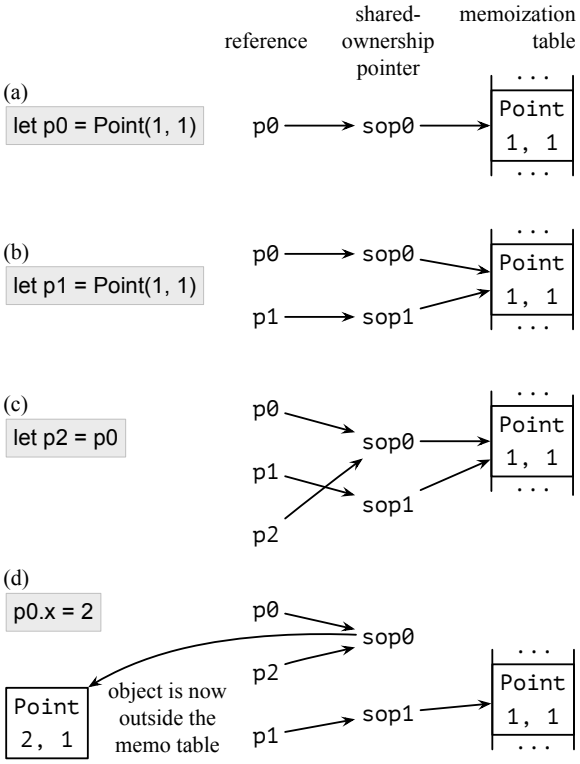


Figure 6. (a) Creation of memoized object. (b) Memoized aliases. (c) Logical aliases. (d) Copy-on-write semantics.

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 memoiazed aliases, the values in the references differ: they point to different shared pointers, but these shared pointers refer to the same object in the memoization table. Example 3.2 shall make this distinction clearer.

Example 3.2. Figure 6 shows some operations performed onto memoized instances of function `Point` (seen in Figure 5). As seen in Figure 6 (a), memoized objects are accessed via shared pointers. Figure 6 (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 6 (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 6 (d) shows.

3.2 Object Creation

Our implementation of Hush creates objects in one of three ways, which Figure 7 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` ← `new`(i) allocates memory to a new object, making `self` a pointer to this newly created location.

$$\frac{\text{o} = \text{new(i)}}{\text{env} \vdash \textsf{self} \leftarrow \text{new(i)} \textbf{ is } \text{env}[\textsf{self}] \leftarrow \text{o}}$$

$$\frac{\text{t} = \text{env}[\textsf{memo}] \qquad \text{t[i]} = \bot \qquad \text{s} = \text{fresh} \qquad \text{o} = \text{new(i)}}{\begin{array}{l}\text{env} \vdash \textsf{self} \leftarrow \textsf{memo new(i)} \textbf{ is } \text{env}[\textsf{self}] \leftarrow \text{s} \\ \qquad\qquad \textbf{and } \text{env}[\text{s}] \leftarrow \text{o} \textbf{ and } \text{t[i]} = \text{o} \\ \qquad\qquad \textbf{and } \textsf{memo} \supset \{\textsf{self}\}\end{array}}$$

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

Figure 7. Rules for creating objects. The environment "env" associates variable names with values.

Creation of objects via memoized functions, e.g., as due to `self` ← `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 entry for i, then a new block of memory o is allocated and stored in t, as seen in the second rule of Figure 7. 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 8 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 8 shows this operation.

$$\frac{\text{o} = \text{env}[\textsf{self}] \qquad f = \text{o[m]}}{\text{env} \vdash \textsf{self.m(i)} \textbf{ is } f\text{(self, i)}} \; \textsf{self} \notin \textsf{memo}$$

$$\frac{\text{s} = \text{env}[\textsf{self}] \qquad \text{o} = \text{env}[\text{s}] \qquad f = \text{o[m]}}{\text{env} \vdash \textsf{self.m(i)} \textbf{ is } f\text{(self, i)}} \; \textsf{self} \in \textsf{memo}$$

Figure 8. 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 8. 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 7 show these code generation rules.

3.4 The Copy-On-Write Semantics

According to the first rule in Figure 9, 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.

$$\frac{\text{o} = \text{env}[\textsf{self}]}{\text{env} \vdash \textsf{self.p} \leftarrow \text{x} \textbf{ is } \text{o[p]} \leftarrow \text{x}} \; \textsf{self} \notin \textsf{memo}$$

$$\frac{\text{s} = \text{env}[\textsf{self}] \qquad \text{o} = \text{env}[\text{s}] \qquad \text{o'} = \text{copy[m]}}{\text{env} \vdash \textsf{self.p} \leftarrow \text{x} \textbf{ is } \text{o'[p]} \leftarrow \text{x} \textbf{ and } \text{env}[\text{s}] \leftarrow \text{o'}} \; \textsf{self} \in \textsf{memo}$$

Figure 9. Rules to modify object properties.

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 9. There are three important facts that must be mentioned:

- 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 `self` do not need to be updated.

4 Formal Properties

The Hush programming language features a mark-and-sweep garbage collector. We have formalized, in the Alloy specification language, some properties that arise from the interaction of memoization and garbage collection. Figure 10 states these properties. For the sake of space, this paper omits the demonstration of the properties that Figure 10 enumerates. However, the extended version of this document[1] contains a full demonstration of each one of them. We use Alloy's SAT solver to validate the proofs. The rest of this section provides an informal overview of this process.

Alloy models sets of states and transitions between these states. Temporal logic statements are validated upon *traces*: exhaustive sequences of states that fully describe the behavior of the system. In our model, transitions between states are provoked by one of the following operations:

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 9. This operation is unsafe if performed

[1] Removed to ensure anonymity

```
P1: Shared pointers can only point to reachable objects
pred p1[] {
  all s: Shared | one s.points => s.points.status = Reachable
}

P2: Objects stored in the memoization table are always reachable
pred p2 [] {
  all o: univ.(memo) | always o.status = Reachable
}

P3: Two objects created via a memoized function with the same
parameters are the same, until one of them is modified
pred p3 [] {
  some s0, s1: Shared | some p: Params | some o0, o1: Object | {
    new[s0, p, o0] and
      (not write[s0, o0] until new[s1, p, o1]) => o0 = o1 }
}

P4: 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: Shared | some p: Params | some o0, o1: Object | {
    new[s0, p, o0] and write[s0, o0] and new[s1, p, o1] => o0 = o1 }
}

P5: Two objects created via a memoized function with the same
parameters are not the same after one of them suffers a copy-on-write
pred p5 [] {
  some s0, s1: Shared | some p: Params | some o0, o1: Object | {
    new[s0, p, o0] and copy_on_write[s0, o0]
    and new[s1, p, o1] => o0' != o1' }
}

P6: Every unreachable object will be eventually collected
pred p6 [] {
  all o: Object | o.status = Unreachable => eventually collect[] and
        o.status' = none
}
```

Figure 10. Properties verified via Alloy.

upon memoized objects, as seen in section 2.3. This operation modifies logical and memoized aliases, as hinted by Property **P4** in Figure 10.

Drop: removes a pointer from scope. The pointed-to object might become unreachable. It will be eventually collected, as per Property **P6** in Figure 10.

Copy-on-Write: copy and modify the state of an object, following the second rule in Figure 9. This operation breaks memoized aliasing relations, as hinted by Property **P5** in Figure 10.

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.

5 Empirical Evaluation

The goal of this section is to evaluate our implementation of memoization in Hush. To this end, we shall provide answers to two research questions:

RQ1: What is the impact of memoization of mutable objects on the running time of programs?

RQ2: What is the impact of hashing on the implementation of memoization.

Benchmarks. 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, 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.

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.

Submarine: 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.

Experimental Setup. Our evaluation runs on an Intel(R) Xeon(R) Silver 4210 2.20GHz processor featuring Linux Ubuntu Server 20.04 (kernel 5.4.0-174-generic). Hush is implemented on Rust v1.77. Running time numbers are the arithmetic measurement of three samples.

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 paper. 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.

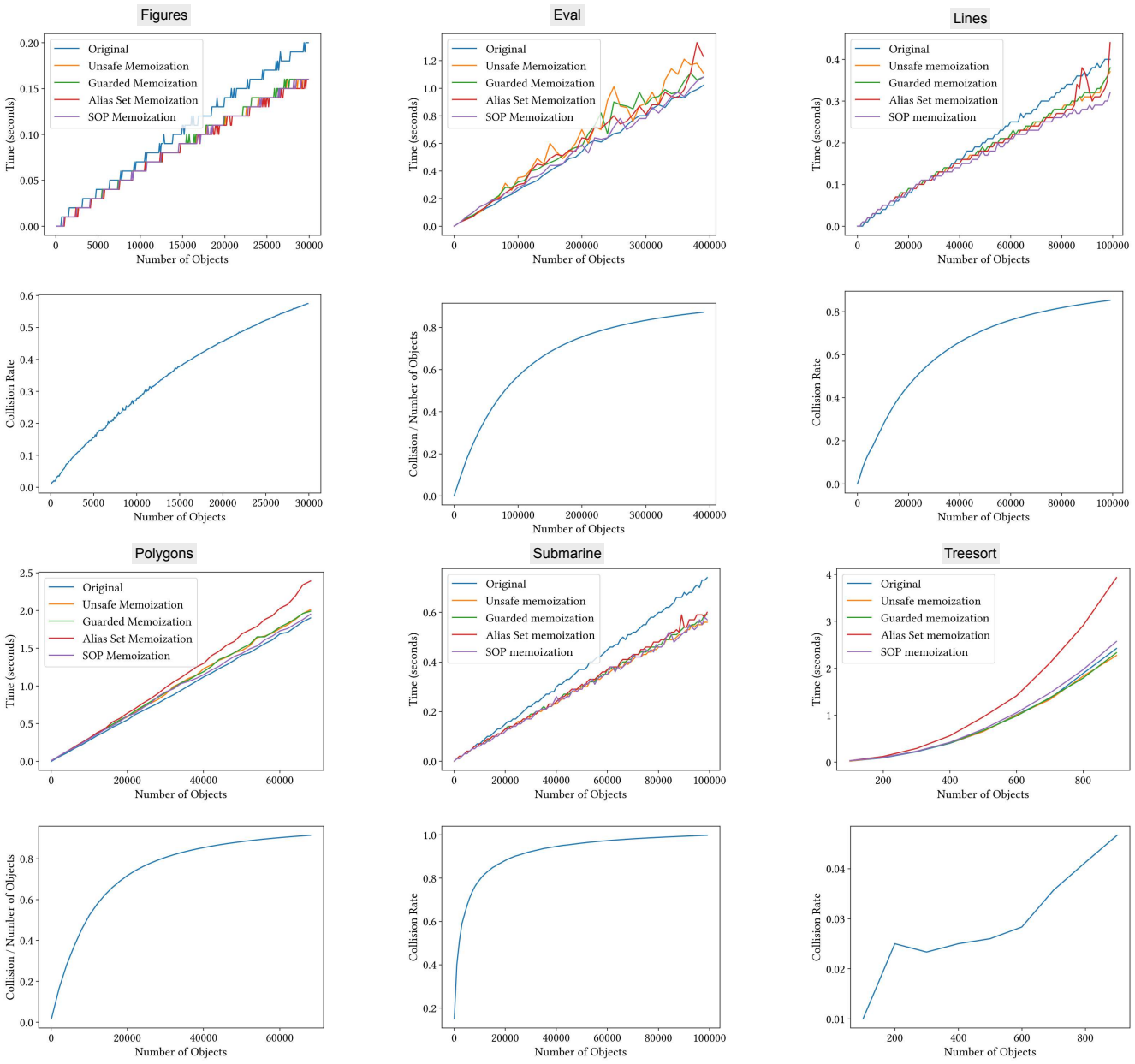


Figure 11. Running time and collision rate of different memoized functions.

Unsafe: a standard implementation of memoization. It might produce wrong programs due to memoization aliasing, as Section 2.3 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 Running Time Impact

Figure 11 shows, for each benchmark, its running time plus the collision rate of memoized objects. Memoization pays off in three benchmarks: `Figures`, `Lines` and `Submarines`. In the latter case, memoization is beneficial because the rate of reused objects is high. Given the nature of the problem, there are only a small quantity of different "Movements" a submarine can use. These movements are memoized objects; hence, more objects are reused from the memoization table. Submarines are also memoized, but they are quickly removed

from the memoization table, due to the copy-on-write semantics explained in Section 3.4. In the case of `Figures` and `Lines`, memoization is advantageous because the functions that construct objects are relatively costly. These constructors receive points as parameters, and use them to create complex objects, such as shapes and three-dimensional lines. Objects are not modified throughout the execution of the benchmarks; thus, they remain in the memoization table.

Nevertheless, all the benchmarks show a similar trend: the cost of our safe approach to memoization is not higher than the cost to implement unsafe memoization. The different between **Sop** and **Unsafe** memoization is often non-statistically significant. Furthermore, there are at least one benchmark: **Polygons**, where **Sop** memoization leads to faster codes (confidence level of 0.95%). We speculate that this extra speed might be the consequence of better cache locality: mutated objects tend to be accessed more often; hence, there are less accesses to the memoization table.

5.2 The Impact of Hashing

In our experience, the main challenge faced when memoizing objects was the impact of *deep hashing*. The comparison between memoized objects involves traversing the graph of values reachable from those objects. In practice, this traversal amounts to 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 might heavily tax the memoization system, as Example 5.1 illustrates.

Example 5.1. The program in Figure 12 creates arithmetic expressions whose size grows exponentially with the value of the input `len`. 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. Memoizing an object involves computing its hash code. The chart in Figure 12 demonstrates that, in this example, memoization time grows exponentially with `len`—a direct consequence of the exponential growth of objects.

```
01 function gen(n)
02   if n == 1 then
03     Num(0)
04   else
05     let T = gen(n-1)
06     Add(T, T)
07   end
08 end
09 len = std.read()
10 gen(len)
```

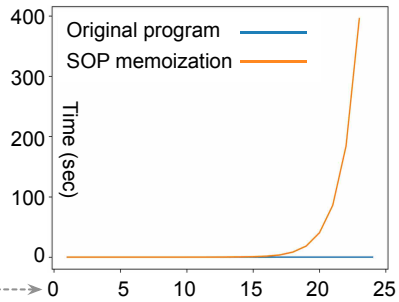


Figure 12. Example that illustrates the heavy impact of deep hashing on the memoization of complex objects.

We have evaluated five different hash functions while implementing memoization in Hush. Figure 13 shows the result of these studies on the **Figures** benchmark. Our best results were observed with the implementation of Fowler-Noll-Vo [8] that is available in the Rust Standard Library. This result is not surprizing: FNV, given its very terse implementatio [12], is well-known for its speed.

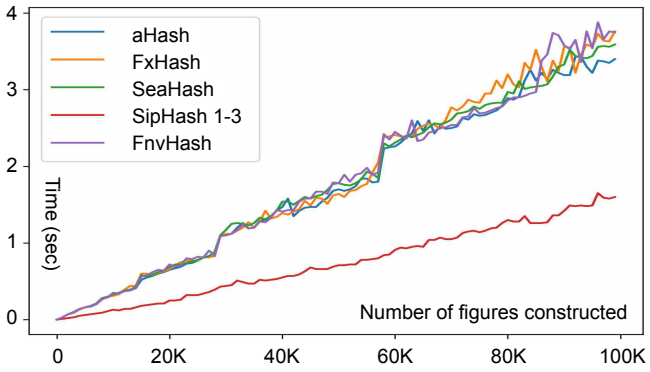


Figure 13. The impact of different hash functions on the running time of the **Figures** benchmark.

6 Related Work

Memoization has been already present in programming languages as early as in 1967 [14]. Indeed, a survey from 1980 shows that memoizing costly function calls has been common practice for most of the seventies [3]. The subject of automatic memoization is more recent. We believe that interest on this topic emerged mostly during the nineties [1, 9, 13, 16]. However, in contrast to this paper, those early studies focused on functions that returned immutable data. To support this statement, we quote Stoffers et al. [19], who is 2016 said that "*till today there is no approach to automated memoization for impure functions*".

The only study of memoization in the context of imperative languages that we know about is due to the work of Stoffers et al. [18, 19]. Stoffers et al. supports memoization of objects, as long as they have no aliases. As soon as dynamic aliasing is detected, Stoffers et al. abort memoization. Again, in their words: "*we conclude that automated memoization with fully unrestricted pointer usage is infeasible as the runtime overhead associated with a dynamic approach would be overwhelming.*" This paper, in contrast, goes one step further: we do allow aliasing of memoized objects, only aborting memoization once mutation is detected.

7 Conclusion

This paper has 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 paper can be adapted to any other programming language that features mutable state. Our version of Hush is publicly available, and 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.

References

[1] Umut A Acar, Guy E Blelloch, and Robert Harper. 2003. Selective memoization. *ACM SIGPLAN Notices* 38, 1 (2003), 14–25.

[2] Gabriel Bastos. 2021. The Hush Programming Manual. https://hush-shell.github.io/. Accessed: 2024-04-25.

[3] R. S. Bird. 1980. Tabulation Techniques for Recursive Programs. *ACM Comput. Surv.* 12, 4 (dec 1980), 403–417. https://doi.org/10.1145/356827.356831

[4] Daniel Bovet and Marco Cesati. 2005. *Understanding The Linux Kernel.* Oreilly & Associates Inc.

[5] Daniel Brown and William R Cook. 2007. *Monadic memoization mixins.* Citeseer.

[6] Paul R. Calder and Mark A. Linton. 1990. Glyphs: flyweight objects for user interfaces. In *UIST* (Snowbird, Utah, USA). Association for Computing Machinery, New York, NY, USA, 92–101. https://doi.org/10.1145/97924.97935

[7] Ole-Johan Dahl and Kristen Nygaard. 1966. SIMULA: an ALGOL-based simulation language. *Commun. ACM* 9, 9 (sep 1966), 671–678. https://doi.org/10.1145/365813.365819

[8] Glenn Fowler, Landon Curt Noll, and Kiem-Phong Vo. 1994. FNV hash history. http://www.isthe.com/chongo/tech/comp/fnv/index.html. Accessed: 2024-03-21.

[9] Marty Hall and J Paul McNamee. 1997. Improving software performance with automatic memoization. *Johns Hopkins APL Technical Digest* 18, 2 (1997), 255.

[10] Ellis Horowitz and Sartaj Sahni. 1974. Computing Partitions with Applications to the Knapsack Problem. *J. ACM* 21, 2 (apr 1974), 277–292. https://doi.org/10.1145/321812.321823

[11] Daniel Jackson. 2019. Alloy: a language and tool for exploring software designs. *Commun. ACM* 62, 9 (aug 2019), 66–76. https://doi.org/10.1145/3338843

[12] Jakub Jelinek. 2024. Implementation of the FNV hash in the Standard Template Library. https://github.com/gcc-mirror/gcc/blob/ee0717da1eb5dc5d17dcd0b35c88c99281385280. File /libstdc++-v3/libsupc++/hash_bytes.cc, Line 123, Accessed: 2024-03-21.

[13] James Mayfield, Tim Finin, and Marty Hall. 1995. Using automatic memoization as a software engineering tool in real-world AI systems. In *AIAI.* IEEE, 87–93.

[14] Donald Michie. 1967. *Memo Functions and the POP-2 Language.* Technical Report. Stanford.

[15] Donald Michie. 1968. Memo Functions and Machine Learning. *Nature* 218, 5136 (1968), 19–22.

[16] Peter Norvig. 1991. Techniques for automatic memoization with applications to context-free parsing. *Comput. Linguist.* 17, 1 (mar 1991), 91–98.

[17] Harald Søndergaard and Peter Sestoft. 1990. Referential transparency, definiteness and unfoldability. *Acta Informatica* 27 (1990), 505–517.

[18] Mirko Stoffers, Daniel Schemmel, Oscar Soria Dustmann, and Klaus Wehrle. 2018. On Automated Memoization in the Field of Simulation Parameter Studies. *ACM Trans. Model. Comput. Simul.* 28, 4, Article 26 (sep 2018), 25 pages. https://doi.org/10.1145/3186316

[19] Mirko Stoffers, Daniel Schemmel, Oscar Soria Dustmann, and Klaus Wehrle. 2016. Automated Memoization for Parameter Studies Implemented in Impure Languages. In *SIGSIM-PADS* (Banff, Alberta, Canada). Association for Computing Machinery, New York, NY, USA, 221–232. https://doi.org/10.1145/2901378.2901386

[20] Gurram Sunitha, Arman Abouali, Mohammad Gouse Galety, and AV Sriharsha. 2023. Dynamic Programming With Python. In *Advanced Applications of Python Data Structures and Algorithms.* IGI Global, 102–122.

[21] David Svoboda and Lutz Wrage. 2014. Pointer ownership model. In *HICSS.* IEEE, 5090–5099.