

Normalization by Origin, Not by Shape

Marco Servetto

marco.servetto@ecs.vuw.ac.nz
Victoria University of Wellington
Wellington, New Zealand

Colin S. Gordon

csgordon@drexel.edu
Drexel University
Philadelphia, USA

Abstract

Memoization is about avoiding recomputation, by caching results and retrieving them when previously-seen inputs are provided again in the future. In the literature, many different ways have been proposed to efficiently (and correctly) check if such input has indeed already been seen.

Prior work largely focuses on structural equality of the keys. In many commonly-used approaches, like PYTHON's `@cache` avoiding (or accepting) observable semantic changes is the responsibility of the programmer.

Here we propose a new approach, where cache keys are considered equal if they are created in the same way. We implement our approach on top of Fearless, a language with reference and object capabilities allowing to control side effects and enforcing determinism of localized code expressions. Our technique guarantees language-enforced unobservable caching: removing the caching annotations does not change the observable program semantic.

CCS Concepts: • Software and its engineering → General programming languages; • Theory of computation → Program constructs.

Keywords: Function memoization, function caching, reference capabilities

ACM Reference Format:

Marco Servetto and Colin S. Gordon. 2026. Normalization by Origin, Not by Shape. In *Proceedings of the 2026 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! '26)*, October 4–9, 2026, Oakland, CA, USA. ACM, New York, NY, USA, 14 pages. <https://doi.org/10.1145/3840586.3843208>

1 Introduction

Memory capacities of general-purpose computers are continually increasing, and caching the results of expensive functions (when safe) can yield substantial speed-ups. The classic example is memoizing a naïve Fibonacci calculation,

which is reduced from exponential time to linear. Many languages have support for memoization. For example, PYTHON offers the *decorator* `@cache`.

```
from functools import cache
@cache
def fib(n):
    if n < 2:
        return n
    return fib(n - 1) + fib(n - 2)
```

If we remove the *decorator* `@cache`, `fib` takes exponential time, while with the decorator `@cache`, `fib` takes (effectively¹) linear time. The decorator (conceptually) expands to the following code:

```
fib_cache = {} # A global cache (dictionary) for fib
missing = object() # A sentinel object

def fib(n):
    # See if a previous call computed the result for n
    res = fib_cache.get(n, missing)
    if res is not missing: # previous result found
        return res # return previous result
    if n < 2:
        res = n
    else:
        res = fib(n - 1) + fib(n - 2)
    fib_cache[n] = res # Add new result to cache
    return res
```

That is, an early exit condition is applied to the start of the function: if the element has been computed previously, we return the cached result; otherwise we compute normally and cache the result on exit. Other interesting examples include obtaining efficient parsers from naïve parsers [33, 38] and accelerating various data structure operations [8].

This form of memoization requires using the function inputs as keys in a hashmap. It works well in this example because there is a single input that is most likely a small number; and small numbers make for good hashmaps keys.

But while the naïve Fibonacci calculation highlights the potential of memoization, there are three major barriers to memoization being both effective (leading to performance gains) and safe (never changing program behaviour beyond performance improvements).

¹PYTHON 3 uses exclusively big-ints, with an interning cache to use a single instance for each small integer, making the analysis slightly more complicated.



This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.

Onward! '26, Oakland, CA, USA

© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2934-8/2026/10

<https://doi.org/10.1145/3840586.3843208>

Problem 1: Large Keys with Slow Comparisons. If the function has many parameters, the hashmap key becomes a tuple of arguments.

```
...
def op(a, b, c, d):
    res = op_cache.get((a, b, c, d), missing)
    ...
```

This makes memoization less effective: while creating the key tuple may be only a minor cost, the operation

```
op_cache.get((a, b, c, d), missing)
```

may be expensive. When we often say that hashmap access is (slow) constant time, we mean with respect to the size of the hashmap. The real cost of `hashmap.get` is on the same order of complexity of the cost of hashing and comparing the map keys. Value-like objects often use structural equality and structural hashing that traverse large parts of their reachable object graph. This makes comparisons, and hashmap operations, scale linearly with the size of the key objects. Even if the map only contains one entry, and that entry is the one we are searching for, accessing the map requires at least one call to `inputKey.hashCode` and `inputKey.equals(existingKeyInMap)`.

That is, when large input is involved, the cost of saving and accessing the input→output connection can exceed the cost of the avoided recomputation.

Problem 2: Observational Instability. Fibonacci is a simple mathematical function, with clear repeatable semantics. Moreover, the input is a (deeply) immutable integer. What about memoization when IO, non-determinism, exceptions, mutation and aliasing come into play? The semantics of PYTHON’s `@cache` are defined as the code expansion above. This optimization is unobservable in cases like our PYTHON Fibonacci; but there is no guarantee that this is the case in general. Adding memoization can cause a *cache-induced semantic change*: the cached program may return an earlier result instead of an updated result that the uncached computation would have produced at that point. Consider caching a function that reads captured mutable state during execution. The cached results would introduce *observational instability* — a situation where caching vs. not caching is observable, and may yield unpredictable changes in program behaviour. Our goal is to provide caching as an optimization with language-enforced unobservability.

Problem 3: Observable Key Equivalence. Ordinary equality (e.g., `.equals` methods) intentionally does not require observable equivalence [51]. It is usually required only to be an equivalence relation: reflexive, symmetric, and transitive. It must also agree with hashing: equal values must have equal hashes. These conditions are enough for predictable hash maps and hash sets, but they intentionally allow many weak equalities. For example, an object may compare by inspecting

only an id field, while ignoring other state that is observable by computations.

For caching, the required strength of equality depends on what the cached computation can observe. If the language merely allows disciplined programmers to write sound caches, then equality only needs to be strong enough for the particular cached computation. Weak equality may be sound for a particular program if the programmer takes responsibility to ensure that the cached computation, and all code reachable from it, never observes the ignored parts of the key or never produces a result dependent on those parts. By contrast, in our example of `.equals` considering only an id field, a computation which does read the other parts of an object and returns varying results depending on those may change behaviour when cached.

In the general case, this means that the programmer would need to consider some common subtle scenarios. Circular object graphs are challenging because defining correct equivalence may require graph isomorphism. More challenging are third party entities: if a key contains an object defined by some other library that we do not control, their comparisons may only respect the weak general comparisons contract, leaving unpredictable observability gaps. The state of such library objects may not be accessible, so we can not even implement our own comparisons for those types.

Instead, a language enforcing unobservable caching must navigate a balance between restricting what a cached expression can observe and enforcing key equality close enough to full observable equivalence between a first computation and return of a memoized result. The weaker the equality check is, the more the language must restrict what cached computations can observe.

Direct observation of object identity (i.e., memory address) is especially destructive: if identity can be observed, then two structurally identical but distinct objects can be distinguished, making general unobservable caching impossible.

Ordinary user-defined equality is sufficient for hashmap correctness, but not for language-enforced unobservable caching. Enforced unobservability requires the language to close the gap: whatever cached code may observe must be covered by automatically checked key equality, and whatever automatically checked key equality does not cover must be made unobservable.

One option is to disentangle those two kinds of equality, and to use a correct by construction system-generated equality when comparing cache keys.

Contributions. In this paper we provide a controlled way to elevate values (of type `T`) to the status of a valid key component (of type `Norm[T]`), normalizing objects by their semantic role in the program rather than by identity or raw structural equivalence. Object identity is not observable by the surface language, and `Norm`s are internally compared by object identity (i.e., current memory address, as updated by

the GC), thus the issues *Large keys with slow comparisons* and *Observable key equivalence* evaporate. Finally, we build our approach on top of Fearless, a language using *reference capabilities* (RCs) and *object capabilities* (OCs) to enforce many safety properties. These features can be used to enforce deterministic behavior for cached expressions/method bodies, in turn preventing *Observational instability*. Overall, our system ensures *unobservable caching*: our memoization is guaranteed to only impact performance. Adding or removing caching does not change the observable program semantics, in that no program can distinguish between a fresh or cached computation results.

Of course, using object identity as an ultimate fine-grained equivalence relation is a well-known approach to solving *Observable key equivalence*; such solutions are only effective when the same actual object is reused instead of a newly created copy. Our controlled way to elevate values to Norms handles this and is a key conceptual novelty of our approach, along with complementary use of other language features to restrict sources of instability.

Sections 2 and 3 provide more background and context on cached functions, and introduce some of the newer language concepts that play an important role in our solution. Section 4 explains how our approach works for caching functions from immutable data to immutable data, and Section 5 extends these ideas to handle caching of computations over mutable data, with reliable cache invalidation. Section 6 further contextualizes our efforts with respect to prior efforts for similar base languages. Because reference capability systems are generally used to enforce data-race-free concurrency, interplay between concurrency and caching is an important consideration. Section 7 considers the role of concurrency in the design of function caching support — both as a technical challenge and as an opportunity for program design. We wrap up discussing future work (Section 8).

2 Background

The standard way to avoid expensive deep equality tests is to replace the deep comparisons with *reference/identity* equality, coupled with an approach to increase the probability that the reference equality check succeeds. Many variants of the latter exist, discussed below. The key idea is that *if* the code systematically uses a single canonical representative in memory for all equivalent structures, then reference equality checks can replace all deep equality checks for (canonicalized) objects used as keys. The extreme version of this is called *maximal sharing* [50], where all structurally equivalent objects (or subobjects) share the same memory allocation and therefore location.

Cached/memoized functions (originally *memo-functions*) were originally proposed by Michie in 1968 [35], and have since enjoyed a long history of reinvention in different contexts. They are, for obvious reasons, more heavily explored

in the context of purely functional languages² where object equality cannot change over time.

The historical crucial building block we rely upon to fight the high cost of key comparisons is Hughes’s insight [29] that effectively splits safety of caching from optimality of caching. By only relying on *reference equality* when looking up in the cache and accepting that two structurally identical structures allocated at different program/code locations would lead to two separate cache entries and repeated computation, we solve both *Large keys with slow comparisons* and *Observable key equivalence*. This is often enough for the particular class of programs that frequently reuse the same exact values. Many programs can be slightly adjusted to improve this reuse, either using custom logic or some language provided constructs, as discussed below.

Obtaining Canonical Representations. Efficacy of memoization improves the more often keys with the same memory address are used. A wide array of techniques can achieve this: interning [16, 40], hash-consing [7, 46], the flyweight pattern [10], and many more. All of these approaches focus on merging all objects with the same internal representation into the same object in memory. While maximal sharing is often the idealized goal, consistently achieving maximal sharing requires either extreme care from developers or direct compiler and runtime support. It also often ends up interning many single-use objects, costing time and space for caching structures that are never reused. This is why most implementations of cached functions today, whether language-supported or as a design pattern, assume *selective*, *opt-in* memoization.

Canonicalization in simple cases. One simple form of canonicalization works for objects/keys whose whole reachable object graph (ROG) is made of immutable objects with canonical constructors that just initialize the state by initializing each field directly with a corresponding constructor parameter. In simple cases, we can use a hashmap to check if an object corresponds to an already-existing key, then either return the previous version, or make a new one and associated it to the key [7, 9]. In this way, if two objects have the same structure in memory, they are also represented by the same pointer in memory from the beginning of their life time. We call this *allocation based* canonicalization: object allocation and canonicalization happen at the same time and can be seen as the same operation. This means that those objects can keep their hashcode/equals implementations as default-/identity based, and other hashmaps and caches would be able to compare them by comparing a single 64-bit address. Even if the reachable object graph may contain thousands

²E.g., HASKELL [12, 29, 42], ACL2 [8], RocQ [9], purely functional fragments of LISP [38] and APL [21]

of sub-objects, comparison is constant and fast. In the simplest cases, this happens at creation time: a duplicated object simply never exists in memory [7, 50].

Crucially, this simple case is not the general case. Our work becomes relevant when the general case comes into play. For example, a common version of the Flyweight pattern requires merging objects with equivalent ROGs; but most code examples showing the Flyweight pattern in action just so happen to only show the simple case we are discussing: there is a single canonical constructor taking a deeply immutable value for each field and directly initializing such fields; and those fields are the intended visible state of the object. Seeing the Flyweight only through the lens of those examples muddles the difference between the Flyweight and the core insight of our work.

Canonicalization in the general case. However, in most OO languages, the situation is much more involved. The state of an object is more loosely connected to the parameters of the constructor/factory, and the observable behavior of an object is even more loosely connected. In many cases objects are first allocated and then initialized via a set of mutations before they become stable enough to be used as keys. This prevents allocation based canonicalization even for immutable data; it is always the case that mutable data cannot be canonicalized, as objects which are temporarily structurally equivalent may change to be inequivalent in the future [48].

Moreover, the very concept of full canonicalization does not play nicely with information hiding. One of the core ideas about OO is that a program can evolve the private state representation of objects as long as the observable behavior stays the same. Consider for example a `Person` type, where the constructor directly stores a string name into a private (final) field. Then, the getter would return `"Unknown"` if the hidden `Person.name` field contains the empty string. In this moment of time, the object `Persons.of("")` and the object `Persons.of("Unknown")` have a different representation (but equal observable semantics³), thus according to full canonicalization, they should be different. Then, in the future, `Person` gets refactored so that `""` is replaced with `"Unknown"` at construction time instead of at getter time. Now full canonicalization would merge the objects `Person.of("")` and `Person.of("Unknown")`. Is this desirable, or is this a violation of encapsulation? With respect to caching, this changes the program's performance characteristics: a cache implemented as an identity hash map over fully canonicalized keys `Map[Person, IdealDog]`, would compute the ideal dog only one time for persons created either with `Persons.of("Unknown")` or `Persons.of("")` in the new program; while in the old one the ideal dog is computed twice. Thus minor refactorings turn into sources of unpredictable performance changes.

³This underlines that *full canonicalization* may still be imprecise: the highest level of reuse would use observable equivalence instead [26], which is however much harder to check than graph isomorphism.

This issue is not as severe when canonicalization is done by the programmer and is the responsibility of the programmer, but it is a real issue if we consider an automatic, language wide canonicalization algorithm, that must work for any kind of object and object graph.

Other barriers to automatic global canonicalization are more practical performance costs. While studies show that programs often contain many equivalent objects [30, 34], identifying them requires expensive traversals of large object graphs. While these traversals can be integrated into garbage collection [7], this is a high tax to pay on all (immutable) object allocations for widely varying gains.

Requiring Use of Canonicalized Versions. Passing a non-canonical input to an API that expects only canonical inputs — can lead to bugs by causing the cheaper equality checks to fail when the more expensive structural equivalence checks would succeed. A natural approach to preventing this problem is to use types to distinguish canonical and possibly-non-canonical versions of structures. Papi et al. [40] and Dietl et al. [16] describe type-qualifier-based extensions to JAVA's type system to ensure that interned strings (or other types) are used in APIs which expect them, introducing types of the form `@Interned T`. Filliâtre and Conchon [19] introduce a hash-consing library for OCAML which uses an abstract type `'a hash_consed` to represent canonicalized values. In these and other cases, these ensure that a value has been canonicalized through *some* cache, but the types do not distinguish between different caches that may exist.

Caching and Mutability. To date all safe approaches to memoized functions or canonicalization rely on using only immutable data [9, 29] or disable canonicalization for structures involving mutable data [7, 48]. In practice most languages permit mutable data, even using it pervasively, so this is not a general solution: arguments to cached functions may be mutated between calls, or the cached function itself may observe other mutated state via captured references. Using function caching or relying on canonicalized representations is therefore unsafe in most settings. We will show that some pieces of ideas from prior work, when combined together with certain *reference capabilities* [11, 22, 25] allows combining the performance insights of prior work with strong yet flexible correctness guarantees.

3 FEARLESS and Capabilities

Our work takes place in the context of FEARLESS [52, 53], an object-oriented (OO) language with *reference capabilities* [11, 22, 25] and *object capabilities* [37] for controlling side effects.

For the examples in this paper, it is sufficient to understand only a small subset of FEARLESS's syntax and types. A simple type declaration in FEARLESS takes forms like

```
Point: {
  .x(): Nat; // abstract methods, as fields
  .y(): Nat;
  .withX(x: Nat): Point -> Points.of(x, this.y());
  .withY(y: Nat): Point -> Points.of(this.x(), y);
}
```

This resembles a class declaration in many languages, and has a few syntactic features of note beyond the usual idiosyncrasies. Method declarations are annotated with a return type on the right, and an `->` separating method signatures from bodies. As with most object-oriented languages, all classes are mutually defined, so the methods of `Point` may reference the companion class which in turn may reference `Point`.

```
Points: { // A way to allocate Points
  .of(x: Nat, y: Nat): Point ->
    { .x -> x; .y -> y }; // Allocate a Point
} // the arguments are captured in method bodies
```

The body of `Points.of` allocates an instance of `Point`; the type `Point` is inferred from the return type of method `.of`. The method signatures are implicitly inherited from the abstract methods of `Point` above.⁴ As usual, we can build upon our `Point` with more user defined types, like `Rectangle` / `Rectangles` below.

```
// Allocate Rectangles and define the type Rectangle
Rectangles: {
  .of(tl: Point, br: Point): Rectangle -> Rectangle: {
    .topLeft(): Point -> tl; // As above, arguments are
    .bottomRight(): Point -> br; // captured in meth bodies
  }
}
```

Here we also show that `Rectangle` (a top level visible type) can be defined directly inside the `.of` factory.

Blending ideas from classic object calculi [13] and modern closure-centric variants of object-orientation [45], FEARLESS lacks explicit fields, whose roles are played (as in classic calculi [3, 31]) by simple closures. Objects in FEARLESS have no constructors, and defining concise factories like `Points.of` is common. Allocation and full initialization are a single operation, that just entails capturing values in the method bodies.⁵ Those factory methods, like `.of`, are just regular methods and thus can perform mutations prior to the allocation, thus there can be no direct connection between the (factory) method parameters and the values captured by the newly created/returned object. The method names here begin with `'` — that is, the dot is part of the name. Method names may in general begin with non-alphabetic characters.

⁴The `{ . . }` notation is technically a multi-lambda [45], though later code examples use it for standard lambdas. As in Java, the actual type of a (multi)lambda is inferred by target-type inference.

⁵Fearless capture is more rigid than, for example, Javascript update. As in Java, it is the value that is captured, not an update-able local variable.

This permits both SMALLTALK [23, 24] style operator overloading and conventions like that in the code above to look like more typical modern object-oriented languages.

When considering observational instability of function memoization, the ability to reason about presence or absence of side effects is key. Most objects in FEARLESS are naturally immutable, and most standard library types (e.g., `Map[K, V]`) are immutable (i.e., functional). FEARLESS does expose a primitive type `Var[T]` which serves a similar purpose to ML reference types, as a mutable container of a single `T`, which can be used for typical imperative structures. To enable improved reasoning about sharing, FEARLESS builds on a long line of work in static *reference capabilities* [11, 22, 25, 49], where reference types are coupled with a type qualifier restricting access through the reference. In particular, reference capability systems allow different aliases to the same object to carry different permissions — so one part of the program may be able to mutate a structure while another part may only be able to observe, but not cause, mutations. Most types in FEARLESS are **imm** (utable) by default, if not otherwise annotated. Updating a `Var[T]` requires holding a **mut** (able) reference of type **mut** `Var[T]`. But the permission restrictions imposed by these capabilities are deeply transitive. A **read** (-only) reference to a container, such as a **read** `Var[mut Var[Str]]` may not be used to obtain a **mut** reference to the inner `Var`. Note that **mut** and **imm** are mutually exclusive, as **imm** implies the reachable objects are considered immutable by the whole program, while **mut** allows mutation. These permissions also constrain what is captured by closures — an **imm** closure capability means the closure does not have access to any mutable state.

This includes global state; like other languages in with reference capabilities, all global state is immutable, so an **imm** closure has no way to cause externally-visible side effects or observe them. This is key to *creating* immutable structures. Like earlier reference capability systems, any code that accepts only immutable inputs may have its result automatically converted to immutable by the type system, even if the construction process included mutating objects and returning mutable objects. If all inputs to a specific method call location are **imm**, and there is no global mutable state, those locally-mutated objects never escape, and are either discarded or can be frozen along with a return value that reaches them. This is also how FEARLESS can create cyclic immutable data: if inputs to the construction are immutable, the creation logic can create the structure as initially immutable, and then the type system can convert the single returned object's capability from **mut** to **imm**, freezing the reachable object graph of the return value. This type system feature is insensitive to the existence of cycles.

Other sources of observational instability include non-determinism from random number generation, I/O, and similar external behaviours. FEARLESS adopts the notion of *object capability* [37] to manage this. Access to any of those or similar platform functionalities is mediated by distinct objects

passed to the program in restricted ways. Notably, all uses of those non-deterministic features require calling `mut` methods — those which require a `mut` capability to the receiver. Thus an `imm` or `read` closure cannot capture or use object capabilities. The details of FEARLESS’s object capabilities will not play a prominent role in the following, but the fact that reference capabilities can restrict use of object capabilities is important to our safety arguments.

4 Our Design

We propose here a concrete design offering normalization via factory/constructor output and memoization built on top of it. We have implemented this for the language FEARLESS, as described in Section 3. FEARLESS also guarantees that any computation that accepts no mutable reference is deterministic.

We implement unobservable caching: a cached expression/method should return an object that is observably equivalent to the object produced by re-executing the expression/method. A valid but pointless way to obtain this behavior would be to simply re-run the expression/method every time it is called, which is unobservable but wasteful. The goal is then to make use of normalization/memoization when conservatively sound, and to re-execute otherwise.

Our strategy is to use only Norms — normalized versions of structures — as keys, and to ground the identity of those keys not in the structural shape of the normed object (slow and complex) but to previously Normed results from a factory for objects.

To clarify; the conventional approach, that we can call *Normalization by shape* asks: “Are these two objects structurally isomorphic?” Instead, our approach *Normalization by origin* asks: “Were these objects produced by the same computation from reference-identical normalized inputs?”

This is our main contribution: that factory/constructor arguments are the right abstraction boundary, not the actual object graph resulting from the factory. Supporting this is the careful integration of earlier ideas from both function memoization and control over side effects in a new complementary blend.

Here by factory we refer to the expression / method invocation that produces an object considered fully initialized and suitable to be an immutable key for caching. This method invocation can internally allocate and edit a complex graph of mutable objects, which are guaranteed by the FEARLESS type system to be fully encapsulated in the resulting value.

This does mean that we accept a form of partial canonicalization instead of maximal sharing: two structurally identical objects may remain duplicated in memory and appear different to caches. We use a wrapper type, `Norm[T]` whose instances represent the normalized version of a `T`. A developer

cannot create instances of `Norm` directly, but must instead use trusted language-provided APIs in the base library.⁶

Operationally, this allows our caches to blend fast comparisons in cache lookups (Hughes’s approach [29]) with type-based constraints ensuring that inputs are canonicalized [16, 19, 40], thus increasing the odds of cache hits. But while this combination is new, the real benefits arise from the details of our interface to canonicalizing data, producing Norms.

Many types from base (the standard library) have a method `.norm` that returns the normalized version of that value.⁷

```
23.norm ⇒ Norm[Nat]{23}
"Hello".norm ⇒ Norm[Str>{"Hello"}
```

Conceptually, those calls access a `HashMap[Nat, Norm[Nat]]` and an `HashMap[Str, Norm[Str]]` hidden in the static data of the standard library. Of course, we said *conceptually*. There is no reason to actually materialize a hashmap from 64-bit numbers to wrapper object of each number. Numbers are small enough that comparing them by structural equality or by object identity makes no difference; but of course the second one has practical overhead costs, so for small standard library objects we use their actual representation as key, and only allocate real `Norm` wrappers for larger ones.⁸ Our language hides those details, so that the mental model shown to the programmer is more uniform: a thing can be used as a key if and only if it is of type `Norm[_]`.

We could then use the Cache type to write

```
.shortestPath(
  addressStart: Str, addressEnd: Str): Path =>
  Cache.of(addressStart.norm, addressEnd.norm,
    // This closure's result will be cached for
    // the passed addressStart and addressEnd
    { a1, a2 => VeryComplex.search(a1.get, a2.get) });
  // .get accesses the Str inside Norm[Str]
```

The unobservability of this caching relies on Fearless’s ability to ensure the lambda `{a1, a2->...}` captures zero values.⁹ Calling `.shortestPath` would require at least three hash map accesses; two on strings (from calling `.norm`) and one on a key object with two `Norm[Str]` fields and a lambda. The third

⁶We declared `Norm` to be `Sealed` inside the base package; then only code inside `base` is allowed to directly create `Norm` objects. Similar sealing mechanisms to require factory usage could be used in other languages.

⁷Having these built-in normalizable types is convenient, not necessary.

⁸This is orthogonal to boxing of primitives, but just a matter of value size: for very small strings (an immutable type), even those larger than a couple machine words, it is often cheaper to directly compare the values than to dig through caches. For these and similarly small values, the value is the canonicalization.

⁹FEARLESS has a type `(CaptureFree)` which ensures lambdas implementing it do not capture any state, and thus are implemented as singletons. This is a common optimization that also happens in `JAVA` or `C++`: a non capturing lambda is a static constant that is reused over and over, not a new object created every time. This is precious because it means that the identity of the lambda can be used as a stable key for our caches. Otherwise, each execution of the code above would allocate a fresh lambda, and thus not be found in the cache.

one is quite fast (using the identities of the two norms and the lambda¹⁰) and unavoidable; but the first two could be slow on long strings.

A method using norms should often expose them like this:

```
.shortestPath(
  addressStart: Norm[Str], addressEnd: Norm[Str]
): Path ->
  Cache.of(addressStart, addressEnd,
    { a1, a2 -> VeryComplex.search(a1.get, a2.get) });
```

In this way it is clear from the API that caching is used and that addresses should be stored and moved around in their normalized form. This is also why the lambda takes `a1` and `a2` as `Norm[Str]` instead of just `Str`: if `VeryComplex.search` were to internally rely on other cached computation, we could pass the input as `Norm[Str]` instead of extracting the string via `.get`; this would avoid costly renormalization steps.

`Cache.of` is conceptually the same kind of abstraction PYTHON offers with `@cache`, but with the guarantees of unobservable caching. That is, if we peel off the `Cache.of` call, we get the following semantically equivalent code.

```
.shortestPath(
  addressStart: Norm[Str], addressEnd: Norm[Str]
): Path ->
  VeryComplex.search(addressStart.get, addressEnd.get)
```

As a stylistic choice, FEARLESS relies on explicitly passing lambdas instead of code rewriting transformations.

4.1 Memoization for User-Defined Datatypes

For each custom type where we can express a factory method over Norm types, we can produce Norms of that type. We now show a conventional example over Points and Rectangles:

```
NormPoints: {
  .of(x: Norm[Nat], y: Norm[Nat]): Norm[Point] ->
    Cache.ofMemo(x,y, { x1,y1 ->
      Points.of(x1.get, y1.get); });}
NormRectangles: {
  .of(topLeft: Norm[Point], bottomRight: Norm[Point]
): Norm[Rectangle] ->
    Cache.ofMemo(topLeft,bottomRight, { t,b ->
      Rectangles.of(t.get,b.get); });}
```

`Cache.ofMemo` works the same as `Cache.of`, except it additionally mints a new Norm for the result. Methods `Cache.of` and `Cache.ofMemo` are overloaded with many different arities,

¹⁰The global `Cache` object mediates access to the memoization functionality. It internally has a global hashmap from arguments and computations to cached results, which is only indirectly accessible by cache updates/retrievals; the hashmap is not directly accessed by FEARLESS code. In the current implementation there is a global identity hashmap from lambda identity to a cache object, internally using a hashmap from pairs of norms to a `Path`. With more control over the backend code generation (FEARLESS targets the JVM) we could instead simply add a synthetic field to the singleton lambda object containing such cache object, thus avoiding an identity hashmap access.

```
NPoints: {
  .of(x: Norm[Nat], y: Norm[Nat]): Norm[NPoint] ->
    Cache.ofMemo(x,y, { x1,y1 -> NPoint:{
      .x:Norm[Nat] -> x1; //by capturing a Norm[Nat]
      .y:Norm[Nat] -> y1; //using the result of .x/.y
    }}); }//for cache accesses is more efficient
NRectangles: {
  .of(topLeft: Norm[NPoint], bottomRight: Norm[NPoint]
): Norm[NRectangle] ->
    Cache.ofMemo(topLeft,bottomRight,
      {t,b->NRectangle:{
        .topLeft:Norm[NPoint] -> t; //if the points are
        .bottomRight:Norm[NPoint] -> b; //used by other
      }}); }//caches, we can access those caches with
//less indirections by exposing/capturing norm values
```

Figure 1. An example demonstrating maximal sharing by using Norm for internal representations.

including zero.¹¹ To cache a long computation with zero arguments,

`Cache.of({computationExpression})` suffices. Or to additionally mint a new norm for custom datatypes, the zero-arity overload of `Cache.ofMemo` may be used:

```
Cache.ofMemo({factoryExpression}).
```

Note that data does not need to be designed with Norm in mind. Above `Point` and `Rectangle` are pre-existing user defined types that have no mention of `Cache/Norm`. However, by carefully designing data to be Norm and to aggregate Norm data, a developer can avoid most cache access indirections for cache input keys, as shown in Figure 1. Moreover, by declaring the types `NPoint` and `NRectangle` directly into the call to `Cache.ofMemo(..)`, we establish a single creation point for those types. This obtains maximal sharing for `NPoint` and `NRectangle` assuming that the user code does not contain other uses of `Cache.ofMemo(..)` returning `NPoint`, `NRectangle` or `Nat`; that is `Nat.norm` normalize `Nats` and `NormRectangles.of` and `NormPoints.of` normalize `NRectangles` and `NPoints`. That is, if for a given type there are multiple code locations of `Cache.ofMemo`, the creation lambdas will have distinct identities, and equivalent allocations will not be memoized.

Defining types in this style is not required. In practice whether this is a worth the extra syntactical weight depends on the application. Weaving Norm through declarations is useful when other caches using those types as keys are common.

Here `NPoint` and `NRectangle` directly store normed values, and the only factory for `NPoint/NRectangle` is protected under a `Cache.ofMemo` lambda.

¹¹In fearless, a zero arg lambda does omit the `->` symbol, so we write `{x+y}` instead of, for example, `{() -> x+y}`

Norms built using this pattern will indeed be maximally shared, and cached expressions over them will only run when truly novel input is provided, returning the cached result in all the other cases.¹²

While this is the highest-performance case, this is also the case that existing approaches to canonicalization support just fine.

Next we defend our claim that our approach degrades gracefully when we get out of this ideal scenario.

4.2 Cyclic Structures

Let us consider some cases involving cyclic object graphs. From a string containing a regular expression, we can create a state machine with circular links between the states. Full canonicalization of the state machine instance would be very challenging, but with our approach, the conventional `Regex.compile(String): RegexStateMachine` functionality is all we need; we can just wrap around `Cache.ofMemo(normStr, {s->Regex.compile(s.get)})` and it is done.¹³ This will produce `Norm[RegexStateMachine]` objects that can be used with `Norm[Str]` inputs for matching: `Cache.of(normStr, normReg, {s, r-> r.get.match(s.get)})`

The last expression only executes full regular expression matching if needed, and just returns the cached result otherwise. If two different regular expression strings are compiled to identical `RegexStateMachine`, our approach would not recognize them as identical and re-run the regex match computation.

Another example could be module dependencies for a build system, where modules can have circular dependencies. Now, assuming a method `Dependencies.of(module: Str): Graph`, we can easily memo-cache the call to `Dependencies.of` and get a `Norm[Graph]` without the need to know anything about how the dependencies are computed. Again, our approach requires us to accept that if some modules have equivalent Graphs, that call would return a duplicated `Norm`, that in turn can cause some spurious re-computation. But the user of our model can mitigate this problem if they apply additional insights from their problem domain. For example, we can imagine that many different modules may have zero dependencies, and it would be better to return a truly normalized empty `Norm[Graph]` representing zero dependencies. This can easily be done, but requires extra code and knowledge of the domain logic. Assuming we have a method `Dependencies.empty`

returning the constant empty dependency graph, it could look something like this¹⁴:

```
.normDep(s: Norm[Str]): Norm[Graph]->
  this._normalizeEmpty(Cache.ofMemo(s, {s1->
    Dependencies.of(s1.get)}));
._normalizeEmpty(d: Norm[Graph]): Norm[Graph]->
  d.get.isEmpty ?{
    .then->Cache.ofMemo({Dependencies.empty});
    .else->d;
  };
```

At a first look, it may seem like `.then->Cache.ofMemo({Dependencies.empty})` is going to compute a new norm every time it is called, but since it is cached over zero parameters, this is actually returning a constant value computed only the first time and then cached.

4.3 Memoized Maps

A very interesting example is the one of actual hashmaps, with arbitrary user-defined logic to compute key equality. If a hashmap is immutable¹⁵ and the keys are large and slow to compare, but we tend to reuse the same key values over and over, we can simply cache the hashmap `.get` method.

```
MemoCache[K, E]: {
  .map: Norm[Map[K, E]];
  .get(key: Norm[K]): E ->
    Cache.of(key, this.map, { k, m -> m.get.get(k.get)})
}
```

This data type wraps a normalized `Map[K, E]` and caches access to the map elements. If we can store the keys as `Norm[K]`, when the cache is warm this map is now as fast as an identity-based hash map while keeping the standard hashmap semantics of performing complex equality checks on the large and unpredictable object graph of the key objects.

4.4 Antipatterns

No form of function memoization is immune to misuse, including ours. But we provide safety guarantees even in the presence of misuse. Our approach, even when used poorly, still guarantees that the caching is unobservable by program logic, though particularly poor usage may nullify most of the performance advantages of our model. Consider the following *bad* code: `Cache.ofMemo({Points.of(12, 43)})`. This mints a new `Norm` for the point (12, 43), but calling `NormPoints.of(12.norm(), 43.norm())` would produce a different `Norm[Point]`. Once this code is in a program, `Cache.of(someNormPoint, ..)` may end up running twice depending on which code is producing the specific point.

¹²Note that unlike other approaches which can achieve maximal sharing [19], this approach is not limited to data allocated directly as immutable. Our example directly builds the `NRectangle`, but arbitrary code can be placed in that expression. Thanks to the presence of *promotion* features [11, 22, 25] in FEARLESS, initially-mutable objects can be mutated arbitrarily before being frozen into `imm`.

¹³Note that the `Regex` library does not need to be aware of the caching.

¹⁴The `myBool ? { .then..; .else..; }` syntax is the FEARLESS way to do conditionals.

¹⁵FEARLESS's `Map[K, E]` type is shallow immutable in the sense that adding, removing, or updating entries results in a new object, as common in functional programming [39]; keys are required to be immutable but the map may contain mutable or immutable elements.

We are not sure if this is a problem. The fact that bad code will run slowly is not news; however, our system has a very *automatic* feel, so we worry that users may not realize that there is still some requirement to follow when using our system. When it comes to larger, long lived systems, having duplicated norms may actually be not a curse, but a blessing in disguise: each lambda passed to `Cache.of` and `Cache.ofMemo` induce a new physical storage location for cached data. Those storage locations can then be tuned or flushed independently by the programmer. Or a library may decide to use its own `Cache.ofMemo` and its own internally minted Norms for its own internal use so that it can control when and how to flush those caches. Such programmatic tuning of caches has long been a useful feature [4, 43] for memoization.

5 Repr for Caches on Mutable Objects

So far we have only addressed computations that consume deeply immutable objects and produce deeply immutable results, with no visible side effects. With some extension, our technique does support mutable objects.

FEARLESS offers the type `Repr[T]`. An instance of `mut Repr[T]` is a safe container for a fully encapsulated `mut T` instance. That is, all mutable objects reachable from this instance are *only* reachable via the `Repr[T]` instance. `Repr` never leaks the `T` instance itself, but has methods `.look(f)` and `.mutate(f)`. They allow the user to observe and mutate the content of the `Repr[T]` by passing closures to make observations or updates. `.look` accepts a lambda taking a read-only view of `T` (`read T`) and returning any immutable `K` observation. Similarly, `.mutate` can receive a lambda accepting the actual `mut T` reference and returning any immutable `K` observation after possibly mutating the `T`. As with the FEARLESS cache APIs discussed earlier, these closures are not permitted to capture mutable state (so `.mutate` cannot leak the underlying reference). It is possible to design a cache system around this set up:

```
repr.cachableLook({content->expression})
```

The code above takes a `repr`, and in a way similar to `.look`, passes the read content to the (non capturing) lambda. `Reprs` make sense as mutable objects, so the identity of the specific `repr` object and the non capturing lambda can be used as the keys in a hashmap connecting to the result.¹⁶ Then, when a call to `Repr.mutate` happens, the caches for that `repr` need to be flushed.¹⁷

Example: editable cached hash maps. In FEARLESS the `Map` type is shallow immutable: the key-element associations are frozen at map creation regardless of key and value types.

¹⁶Again, with better control on the generated code, we could add a synthetic field to `Repr` to store its cache.

¹⁷If the `repr` has a synthetic field with its cache, this is trivial. Another option is to use a Layered approach, with a static `Map[Repr, Map[Lambda, Result]]`.

The type `EMap` (editable map) instead allows for mutation. If `repr` is of type `mut Repr[EMap[K, E]]`, a program may call `repr.cachableLook(normK, {m, k->m.get(k.get)})` to cache the hashmap access, similar to the earlier `Map` example. When `repr.mutate(..)` is called, the cached accesses of the contained `EMap` are flushed. For an `EMap` that receives significantly more reads than writes, this can improve performance.

The two different ways to handle hash maps we have presented weaken, if not remove, the need for an actual identity hash map [15] type to be exposed in the FEARLESS standard library. This is indeed impossible in fearless: FEARLESS does not expose object identities. This is necessary for the determinism guarantees that FEARLESS provides, which is important for far more than just caching.

Subtly, exposing object identities even to hashmaps is unsafe. We initially hoped that by using Norms as keys we could get identity hashmaps that still keep identities unobservable. However, as soon as a cache of a `Cache.ofMemo` is flushed, we could get into both strange behaviors and ways to observe identities. We advice against it and instead we recommend the two map approaches shown before.

Example: cached IDE queries over a mutable project.

A typical example is an IDE or compiler server. The project state is naturally mutable: files are edited, parse trees are updated, indexes are rebuilt, and intermediate data structures are maintained incrementally. At the same time, many IDE operations are read-only observations of that mutable project: diagnostics for a file, completions at a position, the type of an expression or the public API of a package.

Such queries are often expensive, and the same query may be requested many times between edits. For example, hovering, semantic highlighting, quick-fix menus, and completion ranking may all ask for information derived from the same file and position (notably, not the file currently being edited). With `Repr`, the project can remain mutable while still supporting sound caching:

```
project.cachableLook(path.norm, position.norm,
  { p, pathN, posN -> p.completionsAt(pathN, posN) })
```

This call uses the identity of `project`, the identity of the non-capturing lambda, and the normalized arguments as the cache key. The result is an immutable observation, such as a list of completion items.

The important point is that the mutable `Project` itself is never leaked. All read-only observations go through `.look` or `.cachableView`, and all mutation goes through `.mutate`:

```
project.mutate({ p -> p.updateFile(path, newText) })
```

Thus `.mutate` is also the invalidation point: after an edit, all cached observations associated with that `Repr[Project]` are flushed. Without the encapsulation provided by `Repr`, an arbitrary alias to the mutable project could mutate it behind the cache, making the cached result unsound. With `Repr`,

cache invalidation follows from the access discipline of the container (a discipline enforced by reference capabilities).

6 Our experience with Normalization by shape.

We have long been interested in combining function caching with reference capabilities to get unobservable caching, given the long-standing use of reference capabilities for enforcing method purity [27, 28]. Prior to working on FEARLESS, we worked on the reference capability language L42. Both FEARLESS and L42 value minimalism. They are languages where most features can be expressed as libraries and the language itself is actually a very small logical core. To this aim FEARLESS uses closures/lambda with flexible and un-intrusive syntax, while L42 relies on meta-circular meta-programming: Most L42 code does not directly encode types, but works as input/output for other code that transforms it and completes it into the right shape, similarly to how the PYTHON @cache decorator we discussed at the beginning takes the code of the method `fib` in input and produces an elaborated/expanded version of it.¹⁸

While the two languages are different in many ways, their approach to reference capabilities is similar, and all of our considerations would also apply to the other RC based languages like PONY [11] and M# [25].

Before figuring out this solution (Normalization by origin) we tried many different implementation strategies in L42 with the established Normalization by shape mindset. Our goal was to have a standard way to normalize all deeply immutable objects, and to keep the same type: no `Norm[Str]`, just `Str`. If the `Str` needs to be normalized, the system could do it under the hood (or so we hoped!) as in various prior approaches discussed in Section 2. Most prior work on *automatic* caching is from the 1990s, and hardware (from CPUs to memory busses to quantities of available memory) has changed drastically in the last 30 years. Instead of megabytes of memory on DEC Alphas and Pentium 4 machines, we have terabytes of memory.

But the natural approaches tried or considered in the past still had massive computational costs when attempted in L42 [20] and required choosing between undesirable options:

- Repeated full traversal of reachable object graphs, which is quite slow [48]. Worse, to ensure all objects are normalized appropriately it is sometimes necessary to run normalization conservatively, when it is not actually needed.
- Adding an additional field to every object to cache the normalized version of that object like a forwarding pointer. This adds a non trivial space overhead, and is

unused for most allocations, unless it can be merged with the forwarding pointer used in copying garbage collectors [7] (a non-starter in the common modern case of targeting the JVM). This is on top of previous arguments and empirical results [4, 7, 43] that canonicalizing *all* immutable objects simply doesn't pay off.

- Merging normalization and garbage collection, so when objects are normalized, all the pointers of that object everywhere get replaced. Normalization must play nicely with garbage collection in any case [42] (including our approach), but this is a major engineering challenge and adds to the costs of traversal.
- Track how references to non-normalized objects flow through the code, to automatically replace them all when the object is finally normalized, akin to some approaches to incremental computation [3]. This is both difficult and dynamically costly.

In some of those alternatives, normalizing an object required normalizing all objects reachable from it, accessing / locking on concurrent hashmaps, and reconstructing the object so that all reachable objects would point to the normalized versions of those original components.

The main optimization offered by most reference capability languages is unobservable (i.e., data-race-free) parallelism. L42 takes this further, making concurrency automatic in most cases, based on what is safe to parallelize according to the reference capabilities' permissions. Our dream was to cleanly integrate this with automatic unobservable caching, so that both optimizations could build on top of each other. By working under the Normalization-by-shape mindset we got the exact opposite; the two optimizations were *fighting each other*. Normalization ended up as a possibly global operation, and required swapping objects with their normalized form while some parallel worker was actively using them. A working version of this would require substantial synchronization overhead, subtle and costly extensions to garbage collection (notably much more expensive read- and write-barriers), or an additional GC-like mechanism reminiscent of capability revocation in CHERI hardware [18].

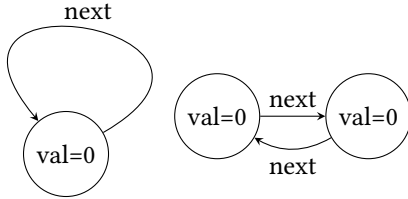
Instead, in our proposed design, the division between `T` and `Norm[T]` distinguishes the few normalized values from all the other ones. The use of the Cache APIs described earlier allows developers to delimit factory boundaries independently of what might be considered traditional object initialization, buying flexibility. We have learned with experience how normalizing all eligible values is still undesirable. In most cases it simply leads to a massive computation time overhead with little memory benefit. Most code will not directly use memoization, and normalizing all values (or even all values of some commonly used types) would be too costly. The design of the `String` type in Java embodies this fact. Strings *can be* interned but this is an optional choice, and most strings

¹⁸More precisely, the L42 meta-programming model is more closely related to F# type providers [41, 47] than to PYTHON decorators, but with extra static guarantees: L42 meta-programming only produce well typed terms when starting from well typed inputs [22, 44].

are not. We now see very clearly why the cost would not be worth the benefit.

As we discussed before, objects can be circular, and there are multiple interpretation of what *equal* even means in the presence of (deeply immutable) circular object graphs.

Are the two object graphs below equivalent or not?



Those could be, for example, nodes of a circular linked list where in one case we have one node with associated value zero and pointing back to itself; in the other case we have two nodes, but it just so happens that they both have associated value zero. If we cannot observe object identity, and those circular linked lists are deeply immutable, there is no way to distinguish those two graphs. L42 considered those two graphs equivalent.

The crucial consideration is that our approach sidesteps this and other subtle questions entirely. If created with the same factory and by the same arguments, two Norms are the same, if not; they are conservatively considered different.

The bright side of L42's graph isomorphism mechanism is that it did allow for automatic, canonical and always correct structural implementations for `toString`, `equals` and `hashCode`, and could have been extended for automatic serialization and deserialization [36].

Given L42's focus on meta-programming, caches where implemented with annotations expanding the code instead of methods taking lambdas. However, L42 code expansion can only create code that would have been possible to directly write in that expanded, thus the expansion had to delegate to publicly accessible primitives. Those primitives were still soundly enforcing unobservable caching, but (purposely) cumbersome to use directly.

The FEARLESS lambda based approach sidesteps those two layers (primitive style and annotation style) offering a single lambda driven caching interface.

7 Parallelism and Contention

There are many computations we already run in parallel for performance reasons where some subproblem solutions are required for multiple other units of work. Compiler analyses, static analyses, graph algorithms, theorem-prover (or other) search, and dynamic programming, are just a few examples.

Caching opens the door for multiple parallel workers to assist each other by pre-filling cache entries which then serendipitously accelerate other tasks, much like a concurrent union-find [6] or the asynchronous convergence of updates used in federated machine learning.

But as soon as multiple workers can execute code at the same time, two problems emerge. The first is traditional contention. We would need to use some form of concurrent hash maps. Then the second is the matter of two workers *competing* to fill a cache. In our prototype, one worker will be first and it will start completing the cache; meanwhile the second worker asking for the cache value would see it as *in progress* and will block waiting for the cache computation to be completed. In turn this can create a deadlock, but only for computations that would have been going in loop/stack overflow in the semantic without caches: Suppose the computation of cache A needs the computation of cache B, that needs the computation of cache A. We may now have two workers, one waiting for A and one waiting for B. More directly, a buggy memoized function may require its own result, and a worker trying to compute it would deadlock on its own similar to a thread trying to complete a future by joining on the result of said future. Note that without multiple workers, any computation trying to access a cache that is currently being filled, would be a clear sign that the semantics without caches would incur a stack overflow.

To prevent those deadlocks from locking threads for an unbound amount of time, we could implement a full dynamic deadlock detection system, but that would require full control of the lifetime and locks of any thread/worker, independently of their relation with caching. For FEARLESS this would be quite difficult, since FEARLESS targets the JVM and relies on the JAVA standard library to implement its parallelism. We opted for a simpler solution: a timeout mechanism; if a worker gets stuck waiting on a cache longer than a certain amount of time it will start to sequentially (re-)compute the result of the cached expression, and flag that cache entry as *timed out* so that any other workers would also compute it sequentially instead of waiting for it. Thanks to determinism, if that computation was doomed to overflow the stack, it will now be able to do so. If this was just a very long computation, we are now paying the price to do the computation twice, as we would have done anyway without the caching model. Note that thanks to the determinism properties that FEARLESS enforces, our simple timeout solution is safe and semantics-preserving.

An even simpler solution would be to just let the multiple workers run in parallel and race to complete, instead of making the other ones wait. This would avoid some locking/synchronization costs, but accepts duplicating computation. While some lock-free algorithms [5, 14] do accept this trade-off and duplicate computation, those algorithms generally do this with short-running computations. Cached computations are user-provided and can be arbitrary long. Moreover, they are exactly computations the user is treating as computationally intensive by choosing to cache them.

Of course, tuning this timeout is not obvious: too high and the application will become unresponsive rather than

crashing, too low and the application will often duplicate computations.

Conceptually, each occurrence of `Cache.of`/`Cache.ofMemo` could have a different timeout, or the timeout could be set as a runtime option when launching the whole app, or each package could have a different timeout. The goal here is to avoid resource exhaustion in the case of long lived applications where an outer controller layer has to run/supervise possibly buggy code that may loop; examples of this could be web servers serving web pages from external customers or test suite runners.

8 Future Work and Open Design Space

Programmers have implicitly absorbed that caches/hash maps need to be used with small keys to be effective, and that caching is a risky endeavor, where it is easy to accidentally introduce bugs for little or no/negative performance gain. Our approach changes the game, and opens new ways of thinking.

8.1 Speculatively Warming Caches

The unobservability of caching implies that reading a cache value and discarding the result is a no-op; but it does fill up the cache. In Section 7 we discussed that concurrent accesses to a cache could accelerate some algorithms. What we did not discuss was that intentionally speculating that a certain computation was likely, and preemptively caching it, may be useful. Instead of just using algorithms which tolerate non-deterministic order of updates, with updates accelerating other threads as a happy secondary effect, this generic mechanism allows us to introduce this behaviour in a lightweight way.

For example, we could add a method `Discard.of({ expression })` taking a lambda capturing only immutable values (an `imm` lambda in FEARLESS) that immediately returns `Void`, but also runs the expression in a separate parallel worker. This is a different way to exploit unobservable caching, and a new tool in the concurrent programming toolbox, where manual software-level explicit speculation is quite difficult and invasive, motivating special-purpose compiler support for that narrow purpose [54]. Here it simply falls out of the fact that caching is guaranteed to be unobservable.

This can assist with parallelizing even algorithms which are largely sequential with highly-irregular shape, by preemptively calling `Discard.of` to solve likely subproblems in parallel with the primary algorithm's logic, which then fills some cache entries before the main algorithm requires those results. Done carefully, it is statistically likely that the final question will be able to use those caches to compute the answer faster. The crucial idea is that during the computation of the final question, we could run code like `NormRectangles`

that can fetch a `Norm[Rectangle]` object that, in turn, is used as a cache key to access a useful precomputed cache.

8.2 Flushing Caches

Another dimension of function caching which is orthogonal to the problems we have discussed, but key to a working system long-term, is managing cache sizes [4, 43]. A short-lived application could simply allow caches to accumulate entries without limits for the duration of the program. But a long-lived application needs to balance performance gains from caching against the memory costs of caches which may accumulate entries for input which are no longer relevant. Currently FEARLESS caches have only a basic cache flush operation, and more detailed APIs remain future work.

Part of the solution typically involves using weak references [17], which hold a reference but are not counted as references by garbage collectors — if the referent of a weak reference has no normal references reachable from standard GC roots, it will be collected and the weak reference will be set to null. A cache may use weak references for keys [42] or values, which ensures that the cache does not prevent garbage collection (e.g., of an input the program is done processing). Reasoning about the performance impacts of this approach is subtle: cache lookups incur an extra indirection from accessing weak references, and some applications may reclaim an object and shortly thereafter allocate a fresh duplicate, leading to cache churn.

Typically frameworks expose ways to tune individual caches [4, 43], since different parts of an application may have different reuse patterns. This requires a way to identify specific caches.

In FEARLESS (as in JAVA) any lambda is conceptually an anonymous inner class; Fearless allows naming them with this syntax:

```
.methName(a:Norm[Str],b:Norm[List[Norm[Str]]]):Nat->
  Cache.of(a,b,Cache7:CacheF[Str,List[Norm[Str]],Nat]({
    aa,bb->Bar.baz(aa,bb)}));
```

Now `Cache7` is the name of that non capturing singleton lambda; by being a singleton its name can be used as the singleton value of that type. Thus that name can be used as a handle to various cache management interfaces.

This is how a FEARLESS program is able to completely empty a cache: `Cache.flush(Cache7)` flushes the cache associated with the closure above. Programmers often know when the cache intensive computation starts and end, so that they can call flush at the end of the computation. This is reasonable with modern memory limits. In the past with only a few hundred megabytes of memory, the concern would have been how to trim the caches *during* the cache intensive computation, because assuming enough memory to just store all the partial works of the computation at hand was a no go. This approach also permits managing each cache individually. Developers could even define multiple lambdas for

the same task in different contexts, if the different use-cases benefit from distinct cache policies.

With the closure acting as the handle for its own cache, we could have operations allowing to set properties of the cache, setting limits (e.g., `Cache.retainLast(2000, lambdaObj)` to retain only the most recent 2000 entries), changing caching strategies (e.g., switching between weak-reference and strong-reference caches), or other adjustments [4, 43]. In some specialized domains, there are even special-purpose caching strategies [32], which a user could provide.

8.3 JIT Time and Compile Time Optimizations

Thanks to FEARLESS's strong type system, the caches are fully unobservable: the program flow can not discover if the caching is happening or not.¹⁹

This means that many different optimizations could be performed by the compiler or the just in time (JIT) compiler. In our vision, the JIT could discover cases of over-caching, where peeling the `Cache.of` call out would improve performance, and dynamically optimize the code by removing such caches. This in turn could incentivise the user to add more `Cache.of` calls, confident that the JIT could remove those 'hints' if appropriate, by reducing the risk of accidentally hurting performance. In general cache information could feed back into the profiling information JIT compilers use to decide when to recompile.

This is similar to what JAVA can do with parallel streams: the system can chose to simply run them sequentially if it seems like a good idea.

9 Conclusions

This paper presented normalization by origin. Unlike interneting and hash-consing, which usually collapse values by structural equality or representation shape, normalization by origin collapses values by factory input, where the code of the factory is conceptually one of the inputs. `Norm` values represent canonicalized versions of actual values and support fast comparison, which allows using the fastest known approaches to memoization lookups. The support for reference capabilities in FEARLESS's type system ensures that the caching behavior remains completely unobservable to the executing program.

FEARLESS is available, including prebuilt binaries [2], and the rest of the language can be learned from an online guide [1].

Acknowledgments

Colin Gordon was supported by NSF Award #CCF-2007582.

References

- [1] 2026. Fearless: Zero to Hero. https://marcoservetto.github.io/ZeroToHero/src/assets/Guide/01_01.html
- [2] 2026. Precompiled binaries of Fearless. <https://github.com/FearlessLang/StandardLibrary/tree/main/fearlessStandalone>
- [3] Martin Abadi, Butler Lampson, and Jean-Jacques Lévy. 1996. Analysis and caching of dependencies. In *Proceedings of the first ACM SIGPLAN international conference on Functional programming*. 83–91. doi:10.1145/232627.232638
- [4] Umut A Acar, Guy E Blelloch, and Robert Harper. 2003. Selective memoization. In *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 14–25. doi:10.1145/604131.604133
- [5] James H Anderson and Mark Moir. 2002. Universal constructions for large objects. *IEEE Transactions on Parallel and Distributed Systems* 10, 12 (2002). doi:10.1109/71.819952
- [6] Richard J Anderson and Heather Woll. 1991. Wait-free parallel algorithms for the union-find problem. In *Proceedings of the twenty-third annual ACM symposium on Theory of computing*. 370–380. doi:10.1145/103418.103458
- [7] Andrew W Appel and Marcelo JR Gonçalves. 1993. *Hash-consing garbage collection*. Technical Report TR-412-93. Princeton University, Department of Computer Science. <https://www.cs.princeton.edu/research/techreps/115>
- [8] Robert S Boyer and Warren A Hunt Jr. 2006. Function memoization and unique object representation for ACL2 functions. In *Proceedings of the sixth international workshop on the ACL2 theorem prover and its applications*. 81–89. doi:10.1145/1217975.1217992
- [9] Thomas Braibant, Jacques-Henri Jourdan, and David Monniaux. 2014. Implementing and reasoning about hash-consed data structures in Coq. *Journal of automated reasoning* 53, 3 (2014), 271–304. doi:10.1007/s10817-014-9306-0
- [10] Paul R Calder and Mark A Linton. 1990. Glyphs: Flyweight objects for user interfaces. In *Proceedings of the 3rd annual ACM SIGGRAPH symposium on User interface software and technology*. 92–101. doi:10.1145/97924.97935
- [11] Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing, and Andy McNeil. 2015. Deny capabilities for safe, fast actors. In *Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control*. 1–12. doi:10.1145/2824815.2824816
- [12] Byron Cook and John Launchbury. 1997. Disposable memo functions. In *Proceedings of the second ACM SIGPLAN international conference on Functional programming*. 310. doi:10.1145/258948.258979
- [13] William R Cook. 2009. On understanding data abstraction, revisited. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*. 557–572. doi:10.1145/1640089.1640133
- [14] Andreia Correia, Pedro Ramalhete, and Pascal Felber. 2020. A wait-free universal construction for large objects. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 102–116. doi:10.1145/3332466.3374523
- [15] Martin De Boer, Stijn De Gouw, Jonas Klamroth, Christian Jung, Matthias Ulbrich, and Alexander Weigl. 2023. Formal specification and verification of JDK's identity hash map implementation. *Formal Aspects of Computing* 35, 3 (2023), 1–26. doi:10.1145/3594729
- [16] Werner Dietl, Stephanie Dietzel, Michael D Ernst, Kivanç Muşlu, and Todd W Schiller. 2011. Building and using pluggable type-checkers. In *Proceedings of the 33rd International Conference on Software Engineering*. 681–690. doi:10.1145/1985793.1985889
- [17] Kevin Donnelly, JJ Hallett, and Assaf Kfoury. 2006. Formal semantics of weak references. In *Proceedings of the 5th international symposium on Memory management*. 126–137. doi:10.1145/1133956.1133974
- [18] Nathaniel Wesley Filardo, Brett F Gutstein, Jonathan Woodruff, Jessica Clarke, Peter Rugg, Brooks Davis, Mark Johnston, Robert Norton, David Chisnall, Simon W Moore, et al. 2024. Cornucopia reloaded: Load barriers for cheri heap temporal safety. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 251–268. doi:10.1145/

¹⁹The user can! Both via observing a different performance and observing effects that are guaranteed to not feed back into the execution model.

- 3620665.3640416
- [19] Jean-Christophe Filliâtre and Sylvain Conchon. 2006. Type-safe modular hash-consing. In *Proceedings of the 2006 Workshop on ML*. 12–19. doi:10.1145/1159876.1159880
 - [20] Isaac Oscar Gariano, Marco Servetto, and Alex Potanin. 2022. Using capabilities for strict runtime invariant checking. *Science of Computer Programming* 224 (2022). doi:10.1016/j.scico.2022.102878
 - [21] Andreas Geyer-Schulz. 1989. Memo. *ACM SIGAPL APL Quote Quad* 20, 2 (1989), 12–27. doi:10.1145/379209.379211
 - [22] Paola Giannini, Marco Servetto, Elena Zucca, and James Cone. 2019. Flexible recovery of uniqueness and immutability. *Theoretical Computer Science* 764 (2019), 145–172. doi:10.1016/j.tcs.2018.09.001
 - [23] Adele Goldberg and Alan Kay. 1976. *Smalltalk-72: Instruction Manual*. Xerox Corporation Palo Alto.
 - [24] Adele Goldberg and David Robson. 1983. *Smalltalk-80: the language and its implementation*.
 - [25] Colin S Gordon, Matthew J Parkinson, Jared Parsons, Aleks Bromfield, and Joe Duffy. 2012. Uniqueness and reference immutability for safe parallelism. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*. 21–40. doi:10.1145/2384616.2384619
 - [26] Peter Grogono and Markku Sakkinen. 2000. Copying and comparing: Problems and solutions. In *European Conference on Object-Oriented Programming*. Springer, 226–250. doi:10.1007/3-540-45102-1_11
 - [27] Wei Huang and Ana Milanova. 2012. ReImInfer: method purity inference for Java. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. 1–4. doi:10.1145/2393596.2393640
 - [28] Wei Huang, Ana Milanova, Werner Dietl, and Michael D Ernst. 2012. ReIm & ReImInfer: Checking and inference of reference immutability and method purity. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*. doi:10.1145/2384616.2384680
 - [29] John Hughes. 1985. Lazy memo-functions. In *Conference on Functional Programming Languages and Computer Architecture*. Springer, 129–146. doi:10.1007/3-540-15975-4_34
 - [30] Alejandro Infante and Alexandre Bergel. 2017. Object equivalence: revisiting object equality profiling (an experience report). In *Proceedings of the 13th ACM SIGPLAN International Symposium on on Dynamic Languages*. 27–38. doi:10.1145/3133841.3133844
 - [31] Samuel Kamin. 1988. Inheritance in Smalltalk-80: A denotational definition. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 80–87. doi:10.1145/73560.73567
 - [32] Hessam Khoshnevisan and Mohamad Afshar. 1996. Space-efficient memo-functions. *Journal of Systems and Software* 35, 1 (1996), 29–41. doi:10.1016/0164-1212(95)00095-X
 - [33] René Leermakers. 1991. Non-deterministic recursive ascent parsing. In *Fifth Conference of the European Chapter of the Association for Computational Linguistics*. <https://aclanthology.org/E91-1012/>
 - [34] Darko Marinov and Robert O’Callahan. 2003. Object equality profiling. In *Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. 313–325. doi:10.1145/949305.949333
 - [35] Donald Michie. 1968. “Memo” functions and machine learning. *Nature* 218, 5136 (1968), 19–22. doi:10.1038/218019a0
 - [36] Heather Miller, Philipp Haller, Eugene Burmako, and Martin Odersky. 2013. Instant pickles: Generating object-oriented pickler combinators for fast and extensible serialization. In *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*. 183–202. doi:10.1145/2509136.2509547
 - [37] Mark Samuel Miller. 2006. *Robust composition: towards a unified approach to access control and concurrency control*. Ph.D. Dissertation. Johns Hopkins University.
 - [38] Peter Norvig. 1991. Techniques for automatic memoization with applications to context-free parsing. *Computational Linguistics* 17, 1 (1991), 91–98. <https://aclanthology.org/J91-1004/>
 - [39] Chris Okasaki. 1998. *Purely functional data structures*. Cambridge University Press.
 - [40] Matthew M Papi, Mahmood Ali, Telmo Luis Correa Jr, Jeff H Perkins, and Michael D Ernst. 2008. Practical pluggable types for Java. In *Proceedings of the 2008 international symposium on Software testing and analysis*. 201–212. doi:10.1145/1390630.1390656
 - [41] Tomas Petricek, Gustavo Guerra, and Don Syme. 2016. Types from data: making structured data first-class citizens in F#. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 477–490. doi:10.1145/2908080.2908115
 - [42] Simon Peyton Jones, Simon Marlow, and Conal Elliott. 1999. Stretching the storage manager: weak pointers and stable names in Haskell. In *Symposium on Implementation and Application of Functional Languages*. Springer, 37–58. doi:10.1007/10722298_3
 - [43] William Pugh. 1988. An improved replacement strategy for function caching. In *Proceedings of the 1988 ACM Conference on LISP and Functional Programming*. 269–276. doi:10.1145/62678.62719
 - [44] Marco Servetto and Elena Zucca. 2010. MetaFJig: a meta-circular composition language for Java-like classes. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*. 464–483. doi:10.1145/1869459.1869498
 - [45] Marco Servetto and Elena Zucca. 2021. λ -based object-oriented programming (pearl). In *35th European Conference on Object-Oriented Programming (ECOOP 2021)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 21–1. doi:10.4230/LIPIcs.ECOOP.2021.21
 - [46] Zhong Shao, Christopher League, and Stefan Monnier. 1998. Implementing typed intermediate languages. In *Proceedings of the third ACM SIGPLAN international conference on Functional programming*. 313–323. doi:10.1145/289423.289460
 - [47] Don Syme, Keith Battocchi, Kenji Takeda, Donna Malayeri, Jomo Fisher, Jack Hu, Tao Liu, Brian McNamara, Daniel Quirk, Matteo Tavecchia, Wonseok Chae, Uladzimir Matsveyeu, and Tomas Petricek. 2012. *Strongly-typed language support for internet-scale information sources*. Technical Report MSF-TR-2012-101. Microsoft Research.
 - [48] Guido Tack, Leif Kornstaedt, and Gert Smolka. 2006. Generic pickling and minimization. *Electronic Notes in Theoretical Computer Science* 148, 2 (2006), 79–103. doi:10.1016/j.entcs.2005.11.041
 - [49] Matthew S Tschantz and Michael D Ernst. 2005. Javari: Adding reference immutability to Java. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. 211–230. doi:10.1145/1094811.1094828
 - [50] Mark van den Brand, Paul Klint, and Pieter Olivier. 1999. Compilation and memory management for ASF+ SDF. In *International Conference on Compiler Construction*. Springer, 198–213. doi:10.1007/978-3-540-49051-7_14
 - [51] Mandana Vaziri, Frank Tip, Stephen Fink, and Julian Dolby. 2007. Declarative object identity using relation types. In *European Conference on Object-Oriented Programming*. Springer, 54–78. doi:10.1007/978-3-540-73589-2_4
 - [52] Nicholas Webster. 2025. *Fearless Automatic Parallelisation*. Ph.D. Dissertation. Open Access Te Herenga Waka-Victoria University of Wellington.
 - [53] Nick Webster, Marco Servetto, and Michael Homer. 2024. The Fearless journey [Draft]. doi:10.48550/arXiv.2405.06233 arXiv preprint.
 - [54] Lingxiang Xiang and Michael Lee Scott. 2013. Compiler aided manual speculation for high performance concurrent data structures. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*. 47–56. doi:10.1145/2442516.2442522

Received 2026-05-15; accepted 2026-06-23