

Normalization by Origin, Not by Shape

Marco Servetto (VUW), Colin S. Gordon (Drexel)



Function Memoization

A classic, *old* ([1968](#)) idea

If you call a function multiple times with the same arguments,
cache, don't recompute

The question is... how do we make this

1. Fast?
2. Safe?

No optimal answer! Lots of trade-offs, let's find a sweet spot...

Unsafe Function Memoization

```
from functools import cache
```

```
@cache
```

```
def fib(n):
```

```
    if n < 2:
```

```
        return n
```

```
    return fib(n - 1) + fib(n - 2)
```

Great for Fibonacci! Exponential $\rightarrow$ Linear!

Unsafe Function Memoization

```
fib_cache = {} # A global cache (dictionary) for fib
missing = object() # A sentinel object
def fib(n):
    res = fib_cache.get(n, missing) # See if a prior call computed result for
n
    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
```

Unsafe Function Memoization

```
from functools import cache
@cache
def fib(n):
    print(f"Calling fib({n})!")
    if n < 2:
        return n
    return fib(n - 1) + fib(n - 2)
```

Now we're memoizing IO...

Are these side effects important?

Harder to say for 3rd-party code...

Three Key Challenges (beyond good caches)

1. Large Keys with Slow Comparisons

- Slow key comparisons make caching slow!

2. Observational Instability

- Caching calls with IO or other side effects *changes the results of the program*

3. Observable Key Equivalence

- Generic equality frameworks (Eq typeclass, Java .equals) let the programmer pick conceptual equivalence
- But functions may be sensitive to differences .equals() ignores

Prior work tackles some of these individually, but not together for imperative code

Reference Capabilities let us combine existing solutions in new ways

Ingredient 1: Reference Capabilities

“Medium-old” (?) idea from [Boyland, Noble, and Retert \(2001\)](#):

Different references to an object act as distinct capabilities

Type system can distinguish read access vs. write access vs...

another quarter-century of variants

We work in the context of FEARLESS, reminiscent of L42, PONY, M#, ...

References carry modifiers: **mut**, **read**, **imm**

(and others we don't need today)

Ingredient 2: Hughes ([1985](#)) Insights

Idea: Only consider two inputs equal if they have the same address

Comparisons are *constant-time*

✓ Fixes Large Keys with Slow Comparisons

For Haskell/Clean-like language: all-immutable data, safe w/o unsafePerformIO

✓ Fixes Observational Instability

Cache hits are *the same object* so memo-equal → observational equivalence

✓ Fixes Observable Key Equivalence

But, some == inputs don't hit in cache... even if allocated at the same line of code

Improving Cache Hit Rates in Memoization Work

- Coalescing via GC (\$\$\$)
- Memoizing on construction (\$\$\$)
- Interning: not automatic, but flexible
 - Semi-implicit (factories)
 - Explicit (interning calls like Java's `String.intern()`)

Our Approach

- Use Hughes' approach for fast cache comparisons
- Use factories to improve hit rates
- Use a type `Norm[ T ]` to further improve hit rates
- Use *reference capabilities* for
 - Key stability (immutable keys)
 - Observational stability (memoize only externally-pure computations)

Norm[T]

- Type-level tag for a normalized T — one from the language-supported memoization APIs (next slide)
- No other way to construct a Norm[T]
- Easy to get the underlying T
- Lets APIs *require* normalized representations
 - (Though multiple factories may exist)
- Lets representations approach maximal sharing (Norm[T] fields)

Note: Any unqualified type in FEARLESS is implicitly **imm**

Origins: `Cache.of(...)` and `Cache.ofMemo(...)`

Special `.norm` method on built-ins

+Factories use two APIs for normalization:

`Cache.of[...](x1:Norm[T1], ..., xn:Norm[Tn], f:F[Norm[T1], ..., T]): T`

`Cache.ofMemo[...](x1:Norm[T1], ..., xn:Norm[Tn], f:F[..., T]): Norm[T]`

Cache.of(...) and Cache.ofMemo(...)

Accept primitives

Normalize for
cache lookup

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

Pull representative for actual computation

Safety & Performance

Cache.of & Cache.ofMemo require

- imm data arguments

- imm lambda expressions (i.e., do not cause external side effects!)

Moreover, the lambda must be CaptureFree

- i.e., represented by function pointer

- lambda identity is stable key for map (vs. allocating every time)

Repr[T]

- So far we've cached immutable keys that look up immutable values
 - With reference capabilities ensuring safety
- We also have a Repr[T]
 - Wraps a *mutable* T
 - .cacheableLook(...) takes read-only observation function
 - .mutate(...) modifies... and clears the cache

Other Things in the Paper

- Simplified handling of cyclic structures
- Cached functional map lookups
- Cached editable map lookups
- Concurrent program acceleration from shared caches

Cool.... Why is this more than shuffling deck chairs?

- The L42 team tried for *years* to get good performance, considering many designs
 - Deep traversals for object equivalence
 - Field overhead for storing normalized version
 - Normalizing during GC
 - Automatically replacing object references with normalized version after construction

So Should We Add RCs Just for Memoization?

No!

Add them for...

- Ownership (ownership & universe types)

- Control over rep-exposure (Javari, OIGJ, L42)

- Data-race-free concurrency (L42, Pony, M#)

And get safe memoization almost free!