Master's Thesis Proposal

Cata Memoization for Generic Data Types

Jort van Gorkum
Computing Science - Programming Technology

Supervisors
Dr. Wouter Swierstra, Dr. Trevor McDonell

February 24, 2022

Todo list

1 Introduction

When computing a result over a data type, a small change would lead to completely recomputing the result. Parts of the recomputation give the same results as the previous computation. By keeping track of the intermediate results and reusing the results when the input is the same, fewer computations have to be performed.

An example of computing a result over a data type is computing the **maximum path sum** over a binary tree. Given a `BinTree`, starting at the root node, find a path from the root node to the leaves that leads to the maximum total.

The implementation of the computation is done in `Haskell`. And the definition of the binary tree and example is:

```
data BinTree = Leaf Int
             | Node BinTree Int BinTree

exampleTree :: BinTree
exampleTree = Node (Node (Leaf 8) 7 (Leaf 9)) 3 (Node (Leaf 5) 4 (Leaf 2))
```

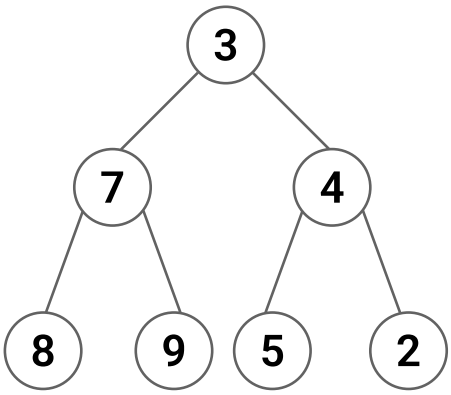


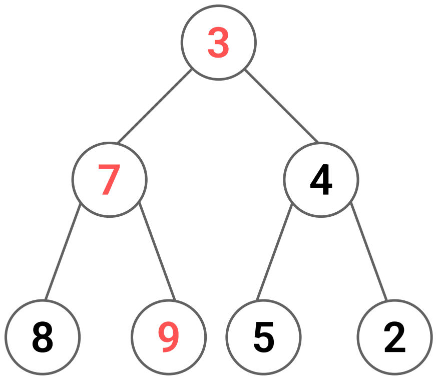
Figure 1: Graphic visualization of the `exampleTree`

An implementation of computing the max path sum over data type `BinTree` is and has a complexity of $\mathcal{O}(N)$.

```
maxPathSum :: BinTree -> Int
maxPathSum (Leaf x)     = x
maxPathSum (Node l x r) = x + max (maxPathSum l, maxPathSum r)
```

And computing the max path sum over the `exampleTree` results in 19.

```
maxPathSum exampleTree ≡ 19
```



Add function complexities to the explanation

To reduce the number of recomputations, first, we need to compare the structure of the data type for equality. When comparing the data type structure for equality, if they are equal, the previously computed result can be reused. Otherwise, the result needs to be recomputed. To compare the structure of the data type in constant time, we introduce the use of hash functions. Generating a hash every time a comparison takes place would be inefficient because part of the structure does not change, which leads to the same hash. Thus, every substructure in the data type stores the hash of its substructure.

In the example, a new data type is introduced: the `MerkleTree`. This data type is the same as the `BinTree`, but the constructors also contain a `Hash`. To create a `MerkleTree`, we traverse through a `BinTree` and hash the structure and store it. Creating a `MerkleTree` has a time complexity of $\mathcal{O}(N)$.

```
data MerkleTree = LeafH Hash Int
                | NodeH Hash MerkleTree Int MerkleTree

merkle :: BinTree -> MerkleTree
merkle (Leaf x)     = LeafH (hash ["Leaf", x]) x
merkle (Node l x r) = NodeH (hash ["Node", x, hl, hr]) l' x r'
  where
    hl = getHash l'
    hr = getHash r'
    l' = merkle l
    r' = merkle r
```

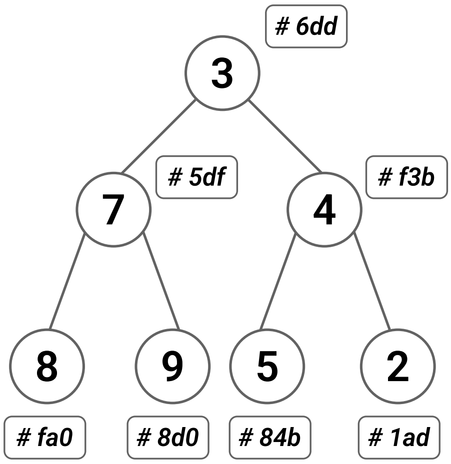
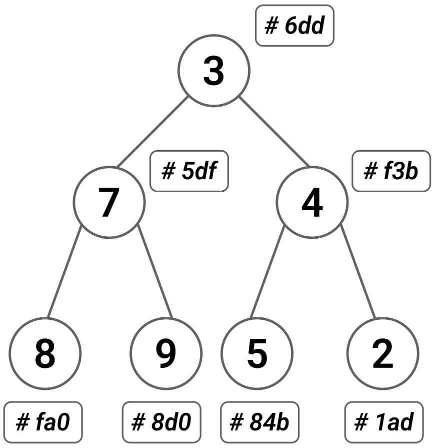


Figure 2: The `MerkleTree` of `exampleTree`

To compute the max path sum and its intermediate results, we can use the precomputed hashes of the `MerkleTree` to efficiently generate a `Map Hash Int`. The complexity of computing the max path sum and the intermediate results is $\mathcal{O}(N)$.

The example implementation of computing the max path sum and its intermediate results can be seen below. However this implementation uses a `union` (`<>`) to combine the `Map`'s which has a complexity of $\mathcal{O}(m * log(n/m + 1)), m <= n$. This can be implemented more efficiently by giving the `Map` to the left side and then to the right side of the Tree, which makes it a constant operation. Except, this would make the code more complex.

```haskell
maxPathSumInc :: MerkleTree -> (Int, Map Hash Int)
maxPathSumInc (LeafH h x)     = (x, insert h x empty)
maxPathSumInc (NodeH h l x r) = (y, insert h y (ml <> mr))
  where
    y = x + max (xl, xr)
    (xl, ml) = maxPathSumInc l
    (xr, mr) = maxPathSumInc r
```



Hash Nodes	Max Sum
# 6dd	19
# 5df	16
# fa0	8
# 8d0	9
# f3b	9
# 84b	5
# 1ad	2

Figure 3: The MerkleTree with intermediate results

When there is a change in the BinTree, only the hashes of the change itself and its parents need to be recomputed. The recomputation of the hash has a time complexity of $\mathcal{O}(M \log N)$ where M is the number of changed nodes.

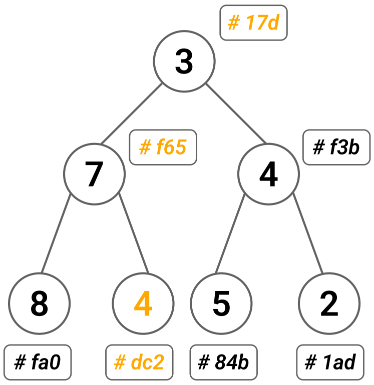


Figure 4: Changed Merkle Tree

Then to compute the max path sum over the Changed Tree, the previously computed Map can be used to reduce the amount of recomputation.

```
maxPathSumMap :: Map Hash Int -> MerkleTree -> (Int, Map Hash Int)
maxPathSumMap m (LeafH h x) = case lookup m h of
  Just y  -> (y, m)
  Nothing -> (x, insert h x m)
maxPathSumMap m (NodeH h l x r) = case lookup m h of
  Just y  -> (y, m)
```

```
  Nothing -> (y, insert h y (ml <> mr))
    where
      y = x + max (xl, xr)
      (xl, ml) = maxPathSumMap m l
      (xr, mr) = maxPathSumMap m r
```

In this paper, an implementation of storing the intermediate results is presented, supporting generic data types. That means the developer only needs to implement the max path sum functionality and the intermediate results are automatically stored.

Function	**Average**	**Upperbound**
`merkle`	$\Theta(N)$	$\mathcal{O}(N)$
`maxPathSumInc`	$\Theta(N)$	$\mathcal{O}(N)$
Change Merkle Tree	$\Theta(M \log N)$	$\mathcal{O}(N)$
`maxPathSumMap`	$\Theta(M \log N)$	$\mathcal{O}(N)$

Function	**Average**	**Upperbound**
`maxPathSum`	$\mathcal{O}(N)$	$\mathcal{O}(N)$

1.1 Contributions

(A) A library needs to be implemented which contains the generic `merkle`, `cataMerkle` and `cataMerkleWithMap` functions

1.2 Research Questions

Using the previously mentioned library, there is a multitude of questions to be answered:

(A) What parameters can be tweaked to have the best ratio of performance and memory usage?

(B) What type of equivalence is needed to reuse the incremental computation?

(C) What type of data structures are the best for storing the incremental computation?

(D) Could the library be used to perform static analysis in a more performant manner?

2 Background

2.1 An Efficient Algorithm for Type-Safe Structural Diffing

The paper *An Efficient Algorithm for Type-Safe Structural Diffing* by Victor Cacciari Miraldo and Wouter Swierstra presents an efficient datatype-generic algorithm called *hdiff* to compute the difference between two values of any algebraic datatype. In particular, the algorithm readily works over the abstract syntax tree (AST) of a programming language[8].

To make the *hdiff* algorithm work, an implementation of which common subtree needs to be defined. The `wcs` function is a function that when given two trees and a subtree, returns the position of the subtree inside the trees if both contain the subtree. Otherwise, the function returns nothing. An example of a naive implementation would be:

```
wcs :: Tree -> Tree -> Tree -> Maybe Int
wcs s d x = elemIndex x (subtrees s ∩ subtrees d)
```

Here the function `subtrees` enumerates all the subtrees of a given tree. Then `elemIndex` returns the index when the subtree is found, otherwise it returns nothing.

The paper identifies two inefficiencies using this naive implementation. (A) Furthermore, enumerating all subtrees is exponential; (B) checking trees for equality is linear in the size of the tree.

To improve the first inefficiency of the naive `wcs` implementation is to use cryptographic hash functions to compare the equality of the trees. To check the trees for equality in constant time the trees are decorated with a hash at every node in the tree. Then, using the precomputed hash and the root node of the given tree, the hash of a subtree is calculated in constant time.

The second inefficiency of the naive `wcs` implementation is improved by using a `Trie`[2] datastructure. Given that a `Hash` is just a `[Char]`, this makes the `Trie` datastructure the preferred choice to store the enumerated subtrees. And because the `Hash` has a constant size the `Trie` lookups are efficient and runs in amortized constant time.

2.2 Sums of Products for Mutually Recursive Datatypes

The paper *Sums of Products for Mutually Recursive Datatypes* written by Victor Cacciari Miraldo and Alejandro Serrano[7] presents a new approach to generic programming using recursive positions to handle mutually recursive families and the *sum-of-products* structure. This work (`generics-msrop`) is later used by the paper *An Efficient Algorithm for Type-Safe Structural Diffing* by Victor Cacciari Miraldo and Wouter Swierstra[8] to define the generic version of their diffing algorithm. Compared to existing generic programming libraries, `generics-mrsop` has *deep explicit recursion, sums of products* and supports *mutually recursive datatypes.*

Explicit recursion There are two ways to represent values. One contains the information on what properties of a datatype are recursive. The other does not contain that information. If we do not know explicitly if the property is recursive, then only one layer of the value can be formed into a generic representation. This is called *shallow* encoding. If we explicitly keep track of the recursive property, then the entire value can be transformed into a generic representation. This is called *deep* encoding. Using the *deep* encoding more datatypes can be defined generically (e.g., a generic *map* or generic Zipper datatype).

Sums of Products The `generic-sop` library uses a list of lists of types. The outer list represents the sum and the inner list represents the product. The sum represents the choice between two constructors; the product represents a combination of two constructors. An example of a `Code` representation of a `BinTree` is

```
data BinTree a = Leaf a
                | Node (BinTree a) (BinTree a)


Code_BinTree(Bin a) = `[`[a], `[Bin a, Bin a]]
```

Here the ` sign in the code promotes the definition to the type-level instead of a run-time value. The use of *Sums of Products* makes it considerably easier to represent generic datatypes.

Mutually recursive datatypes Most of the generic programming libraries are restricted to only allowing recursion on the same datatype, which is the one being defined. Mutually recursive datatypes are recursively defined in each other's terms, meaning that most generic programming libraries do not support mutually recursive datatypes. This limits the ability to generically represent the syntax of many programming languages. Thus `generic-sop` introduces recursive positions on a type level, which can be used to define mutually recursive datatypes.

2.3 Concise, Type-Safe, and Efficient Structural Diffing

The paper *Concise, Type-Safe, and Efficient Structural Diffing* written by Erdweg, Sebastian and Szabó, Tamás and Pacak, André presents a structural diffing algorithm called *truediff* [5]. *truediff* ensures that the patches produces are concise and type safe, and with a performance by an order of magnitude higher than Gumtree[6] and the *hdiff* [8] algorithm.

To compute the difference between a source tree and a target tree, *truediff* operates in four steps: (1) prepare subtree equivalence relations; (2) find reusable candidates; (3) select reusable candidates; (4) and compute the edit script.

The equivalence relations used in step 1, exist out of two equivalence relations, both encoded through cryptographic hashes. The first equivalence relation is used to identify reusable candidates. The second equivalence relation is used to identify preferred reusable candidates. The paper found that using structural equivalence to identify candidates and literal equivalence to select preferred candidates yields very concise edit scripts.

3 Selective Memoization

The paper *Selective Memoization* by Umut A. Acar, Guy E. Blelloch and Robert Harper[1] presents a framework for applying memoization selectively. Also, it describes what type of key issues there are with implementing memoization efficiently: (a) equality; (b) precise dependencies and (c) space management.

For equality, the cost of having an equality test can negate the advantage of using memoization. In the paper, there are a few approaches proposed to alleviate this problem. The first is based on the equality test not having to be exact. So, for expensive equality tests, it could determine to skip the test or use a less expensive equality test.

The second approach suggested is to ensure that there is only one copy of every value, known as a "hash consing". If there is only one copy, equality can then be implemented by comparing locations. The problem with hash consing is it demands a large amount of memory and has trouble working with the garbage collection. An alternative proposed by Pugh and Teitelbaum[10], is lazy structure sharing.

Maybe add an explanation of what lazy structure sharing is

For precise dependencies, to maximize the reuse of the results, the results need to depend on the true dependencies. This means that the results can depend on a subset of the parameters. The subset of parameters leads to a partial equality check, which can increase the likelihood of results reuse.

For space management, as the program gets executed, the size of the space used can become a limiting factor. To alleviate this problem the results should be disposed of when the space usage becomes too large. The disposed of result should be the one that leads to the fewest amount of recomputation. One widely used approach presented in the paper is to replace the least recently used entry. The disposed of policy must be application-specific according to the paper, because there are programs whose performance is made worse, by using a fixed policy.

4 Preliminary Results

The prototype algorithm is written using simpler self-defined generic datatypes with a fixpoint, which are defined in Appendix A and B[1]. An example of how the generic datatypes can be used is:

```
data Tree a = Leaf a
            | Node (Tree a) a (Tree a)

type TreeG a = Fix (TreeF a)
type TreeF a = K a                  -- Leaf
            :+: ((I :*: K a) :*: I) -- Node
```

Using the generic datatypes a `merkle` function can be defined, where at every recursive step of the datatype a `Hash` is stored. To merkelize a datatype, the datatype has to have the `Merkelize` constraint. The `Merkelize` type class is a class containing a single function `merkleIn` which converts the once unpacked `Fix` datatype into a unpacked `Fix` which contains a Hash at every recursive step[2].

```
merkle :: Merkelize f => Fix f -> Fix (f :*: K Hash)
merkle = In . merkleIn . unFix
```

```
class (Functor f) => Merkelize f where
  merkleIn :: (Merkelize g)
           => f (Fix g) -> (f :*: K Hash) (Fix (g :*: K Hash))
```

The generic datatypes can also use a `cata` function. The `cata` or catamorphism is a generalization of the concept of a fold, which means it deconstructs a data structure into its underlying functor[3].

```
cata :: Functor f => (f a -> a) -> Fix f -> a
cata alg t = alg (fmap (cata alg) (unFix t))
```

The `cata` function can then be used to, for example, calculate the sum of all the values of the nodes and the leaves of the tree.

```
cataSum :: TreeG Int -> Int
cataSum = cata (\case
  Inl (K x)                      -> x
  Inr (Pair (Pair (I l, K x), I r)) -> l + x + r)
```

[1] The source code is on GitHub at https://github.com/jortvangorkum/memo-cata

[2] The implementation of the generic datatypes for the `Merkelize` type class can be found in Appendix C.

To keep track of the incremental computation of the summation of the tree, a HashMap[4] is used. The calculation of the incremental step is inserted into the HashMap and a pair of the HashMap and the result is returned. The implementation for the `TreeG` datatype is:

```
cataMerkleTree :: TreeG Int -> (Map Hash Int, Int)
cataMerkleTree t = cata sumTree merkleTree
  where
    merkleTree :: Fix (TreeF a :*: K Hash)
    merkleTree = merkle t

    sumTree :: (TreeG Int :*: K Hash) Int -> Int
    sumTree (Pair (px, K h)) = case px of
      -- Leaf
      Inl (K x)
        -> (M.insert h x M.empty, x)
      -- Node
      Inr (Pair (Pair (I (xl, ml), K x), I (xr, mr)))
        -> let n = x + xl + xr
            in (M.insert h n (ml <> mr), n)
```

Then using the previously generated HashMap, we can then calculate the result reusing the previously incremental computations:

```
cataMerkleTreeWithMap :: Map Hash Int -> TreeG Int -> (Int, Map Hash Int)
cataMerkleTreeWithMap m (In (Pair (x, K h))) =
  case lookup h m of
    Just n  -> (n, m)
    Nothing -> case x of
      Inl (K x) -> (x, insert h x empty)
      Inr (Pair (Pair (I l, K x), I r)) -> (x', m')
        where
          (xl, ml) = cataMerkleTreeWithMap m l
          (xr, mr) = cataMerkleTreeWithMap ml r
          x' = x + xl + xr
          m' = insert h x' mr
```

Using the previously defined `cata` functions we can determine the performance of the functions by using the `criterion`[9] package. For a benchmark in `criterion`, first, the environment is set up. Then the bench function is executed multiple times within a certain timeframe. The result of the multiple executions is used to calculate the mean and standard deviation of the time executed.

The results of the `cataSum`, `cataMerkleTree` and `cataMerkleTreeWithMap` is seen in the graph.

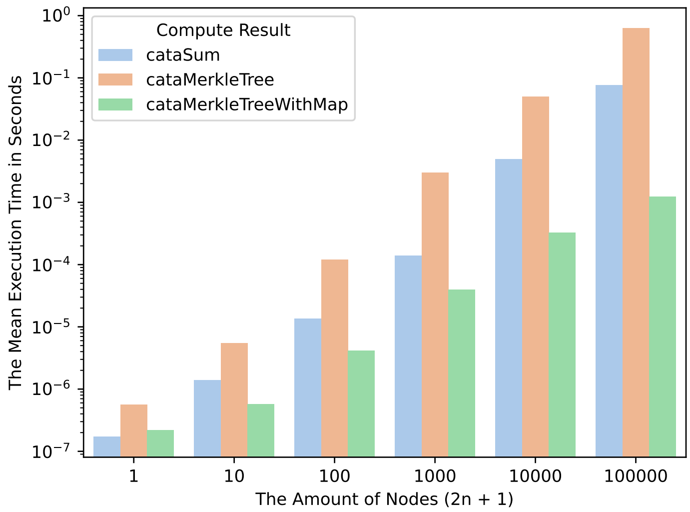


Figure 5: Compute the result

4.1 Future challenges

The problem with this implementation is that it only works for the `TreeG` datatype. The goal would be to create a generic function, where only the `cataSum` would be defined and the result would automatically contain the intermediate results. A generic definition could look something like this:

```
cataMerkle :: (f a -> a) -> Fix (f :*: K Hash) -> State (Map Hash a) a
```

Using the `cataMerkle` function would lead to only needing to implement the `cataSum` function and the intermediate results are then automatically stored.

5 Timetable and Planning

5.1 Exploratory topics

During the first part of the Thesis project, multiple topics are thought of that need further research/implementation in the second part of the Thesis project.

To implement the generic `merkle`, `cata` and `cataWithMap` functions, the `generics-msrop` library described in Section 2.2 is a good candidate to use for implementing these functions. This is because it supports mutually recursive datatypes, meaning that a large group of datatypes are supported.

In the paper *Selective Memoization* in Section 3, there are three key issues highlighted to focus on for implementing an efficient memoization algorithm: equality, precise dependencies, and space management. The parameters for these key issues could be further researched.

In the paper *Concise, Type-Safe, and Efficient Structural Diffing* in Section 2.3 they describe using two equivalence relations instead of one. Using two equivalence relations could lead to more opportunities for reusing computed results. However, further research is needed on how feasible it is using two equivalence relations.

For the data structures used for storing the incremental computation, the easiest to use would be using a `HashMap`. But, the paper *An Efficient Algorithm for Type-Safe Structural Diffing* described in Section 2.1 suggest using a different data structure, the `Trie` data structure. Further research could be done in comparing the performance and memory usage of both data structures.

Add explanation of incremental updating merkle tree and using Zipper or Lenses to solve it

5.2 Schedule

Priority 1 is the lowest, 5 is the highest.

Category	Work	Priority
Implementation	Implementing Generic CataMerkle library using `generics-msrop`	5
	Implementing incremental update merkle tree	5
	Implementing tests	4
Experiments	Creating benchmarks	5
	Experiment using different parameters for space management	4
	Experiment using different data structures	4
	Experiment using real-world data	3
	Experiment using different parameters for equality	2
	Experiment using different parameters for precise dependencies	1

Table 1: Category priority list

Week	Date	Category
Week 1 - 4	28 feb - 25 mar	Implementation
Week 5 - 10	28 mar - 06 may	Experiments
Week 11 - 13	08 mar - 13 may	Writing
Week 14 - 17	16 may - 3 jun	Feedback
Week 18 - 19	6 jun - 17 jun	Time Left (Vacation/Overdue Work)
Week 20	20 jun - 24 jun	Finalize
Week 21	27 jun - 1 jul	Vacation
Week 27	08 aug - 12 aug	Submission
Week 28	15 aug	End Date Research Project

Table 2: Planning per category

6 Appendix

A Definition Generic Datatypes

```haskell
data I r         = I r
data K a r       = K a
data (:+:) f g r = Inl (f r) | Inr (g r)
data (:*:) f g r = Pair (f r, g r)
```

B Definition Fixpoint

```haskell
data Fix f = In { unFix :: f (Fix f) }

instance Eq (f (Fix f)) => Eq (Fix f) where
    f == g = unFix f == unFix g

instance Show (f (Fix f)) => Show (Fix f) where
    show = show . unFix
```

C Implementation Merkelize

```haskell
instance (Show a) => Merkelize (K a) where
  merkleIn (K x) = Pair (K x, K h)
    where
      h = hashConcat [hash "K", hash x]

instance Merkelize I where
  merkleIn (I x) = Pair (I prevX, K h)
    where
      prevX@(In (Pair (_, K ph))) = merkle x
      h = hashConcat [hash "I", ph]

instance (Merkelize f, Merkelize g) => Merkelize (f :+: g) where
  merkleIn (Inl x) = Pair (Inl prevX, K h)
    where
      (Pair (prevX, K ph)) = merkleIn x
      h = hashConcat [hash "Inl", ph]
  merkleIn (Inr x) = Pair (Inr prevX, K h)
```

```
    where
      (Pair (prevX, K ph)) = merkleIn x
      h = hashConcat [hash "Inr", ph]

instance (Merkelize f, Merkelize g) => Merkelize (f :*: g) where
  merkleIn (Pair (x, y)) = Pair (Pair (prevX, prevY), K h)
    where
      (Pair (prevX, K phx)) = merkleIn x
      (Pair (prevY, K phy)) = merkleIn y
      h = hashConcat [hash "Pair", phx, phy]
```

D Results of computing the sum of a Tree

Amount	Action	Mean	Stddev
1	Generate (Result, Map)	5.662e-07	1.195e-08
1	Generate (Result, Map) with Map	2.208e-07	5.237e-09
1	Generate Result	1.713e-07	1.721e-09
10	Generate (Result, Map)	5.456e-06	6.462e-08
10	Generate (Result, Map) with Map	5.744e-07	8.788e-09
10	Generate Result	1.401e-06	1.132e-08
100	Generate (Result, Map)	1.205e-04	2.379e-06
100	Generate (Result, Map) with Map	4.165e-06	6.188e-08
100	Generate Result	1.358e-05	1.826e-07
1000	Generate (Result, Map)	3.024e-03	9.485e-05
1000	Generate (Result, Map) with Map	3.955e-05	6.024e-07
1000	Generate Result	1.387e-04	1.708e-06
10000	Generate (Result, Map)	5.018e-02	2.108e-03
10000	Generate (Result, Map) with Map	3.280e-04	5.800e-06
10000	Generate Result	4.994e-03	1.123e-04
100000	Generate (Result, Map)	6.253e-01	2.174e-02
100000	Generate (Result, Map) with Map	1.228e-03	1.836e-05
100000	Generate Result	7.592e-02	1.505e-03

Table 3: Compute the result

References

[1] Umut A Acar, Guy E Blelloch, and Robert Harper. "Selective memoization". In: *ACM SIGPLAN Notices* 38.1 (2003), pp. 14–25.

[2] Peter Brass. *Advanced data structures.* Vol. 193. Cambridge University Press Cambridge, 2008, pp. 336–356.

[3] *Catamorphisms - HaskellWiki.* URL: `https://wiki.haskell.org/Catamorphisms` (visited on Jan. 26, 2022).

[4] *Data.Map.* URL: `https://hackage.haskell.org/package/containers-0.4.0.0/docs/Data-Map.html` (visited on Feb. 8, 2022).

[5] Sebastian Erdweg, Tamás Szabó, and André Pacak. "Concise, type-safe, and efficient structural diffing". In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation.* 2021, pp. 406–419.

[6] Jean-Rémy Falleri et al. "Fine-grained and accurate source code differencing". In: *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering.* 2014, pp. 313–324.

[7] Victor Cacciari Miraldo and Alejandro Serrano. "Sums of products for mutually recursive datatypes: the appropriationist's view on generic programming". In: *Proceedings of the 3rd ACM SIGPLAN International Workshop on Type-Driven Development.* 2018, pp. 65–77.

[8] Victor Cacciari Miraldo and Wouter Swierstra. "An efficient algorithm for type-safe structural diffing". In: *Proceedings of the ACM on Programming Languages* 3.ICFP (2019), pp. 1–29.

[9] Bryan O'Sullivan. *Criterion: A Haskell microbenchmarking library.* URL: `http://www.serpentine.com/criterion/` (visited on Feb. 8, 2022).

[10] William Pugh and Tim Teitelbaum. "Incremental computation via function caching". In: *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages.* 1989, pp. 315–328.