Computing Science MSc Thesis

# Generic Incremental Computation for Regular Datatypes in Haskell


*Author*
Jort van Gorkum (6142834)

*Supervisors*
*Dr.* Wouter Swierstra
*Dr.* Trevor McDonell

Faculty of Science
Department of Information and Computing Sciences
Programming Technology


August 8, 2022

**Abstract**

Incremental computation is a method which tries to save time by only recomputing the output of changed input. A technique of incremental computation is memoization. Memoization stores the result of a computation and returns the cached result when the same input occurs again. As a result, a large part of memoization becomes dependent on determining if the input is equal to an already cached input. This can become problematic when a computation is given a large recursive data structure. To improve the performance of memoization this paper introduces an incremental algorithm which determines the equality in constant time. This is accomplished by storing hash values/digests, which describe the internal structure, inside the data structure. Furthermore, the incremental algorithm describes how to efficiently update the digests, using a Zipper, when the data structure changes. The incremental algorithm is then implemented using Datatype-generic programming, to support the class of regular datatypes. Meanwhile, the usage of the generic implementation stays the same for the developers as writing the non-incremental algorithm in Haskell. Finally, we show that the performance is better than the non-incremental version with minimal extra memory usage, when correctly tuned with cache policies.

# Contents


**1 Introduction** **4**
1.1 Contributions . . . 7

**2 Specific Implementation** **8**
2.1 Merkle Tree . . . 10
2.2 Zipper . . . 11
2.2.1 Zipper Merkle Tree . . . 12

**3 Datatype-Generic Programming** **13**
3.1 Introduction . . . 13
3.2 Explicit recursion . . . 15
3.3 Sums of Products . . . 16
3.4 Mutually recursive datatypes . . . 17

**4 Generic Implementation** **18**
4.1 Regular . . . 18
4.2 Generic Zipper . . . 21
4.3 Cache Management . . . 22
4.3.1 Cache Addition Policies . . . 22
4.3.2 Cache Replacement Policies . . . 24
4.4 Pattern Synonyms . . . 25

**5 Experiments** **26**
5.1 Method . . . 26
5.2 Results . . . 27
5.2.1 Execution Time . . . 27
5.2.2 Memory Usage . . . 28
5.2.3 Comparison Cache Addition Policies . . . 29

**6 Discussion & Conclusion** **32**
6.1 Related Work . . . 32
6.1.1 Comparison of equality in constant time . . . 32
6.1.2 Storing the cached results . . . 32
6.1.3 Updating the input . . . 33
6.2 Future Work . . . 33
6.2.1 Support for Mutually Recursive Datatypes . . . 33
6.2.2 Implement the incremental algorithm using Sums-of-Products . . . 34
6.2.3 Benchmarking with real-world data . . . 34
6.2.4 Support for a new input without changes . . . 34
6.2.5 Prioritization . . . 34
6.3 Conclusion . . . 35

**A Generic Programming** **36**
A.1 Functor instances for Pattern Functors . . . . . 36

**B Cache Management** **37**
B.1 Implementation Recursion Depth . . . . . 37

**C Results** **38**
C.1 Minimum of 10 recursion depth . . . . . 38
C.2 Individual benchmark results - Worst Case . . . . . 40
C.2.1 Linear trend line . . . . . 40
C.2.2 Logarithmic trend line . . . . . 41
C.3 Individual benchmark results - Average Case . . . . . 43
C.3.1 Linear trend line . . . . . 43
C.3.2 Logarithmic trend line . . . . . 44
C.4 Individual benchmark results - Best Case . . . . . 46
C.4.1 Linear trend line . . . . . 46
C.4.2 Logarithmic trend line . . . . . 47

# Todo list

# 1

# Introduction

Incremental computation is an approach to improve performance by reusing results of a previously computed result when both of the inputs are equal. There are multiple applications where incremental computation has a positive effect: GUIs (e.g., DOM diffing), spreadsheets, attribute grammar evaluation, etc. An example, where incremental computation has significant effect on performance, is computing the Fibonacci sequence.

```
fib :: Int -> Int
fib 0 = 0
fib 1 = 1
fib n = fib (n - 2) + fib (n - 1)
```

**Figure 1.1:** The implementation of the Fibonacci sequence in Haskell.

The implementation of the incremental computation for the Fibonacci sequence is called *memoization*. Memoization is a technique that caches the result of a function and reuse the cached result when the input is the same. A well-used Haskell package which implements memoization is the `MemoTrie` package[7].

```
memoFib :: Integer -> Integer
memoFib = memo memoFib'
  where
    memoFib' 0 = 0
    memoFib' 1 = 1
    memoFib' n = memoFib (n - 2) + memoFib (n - 1)
```

**Figure 1.2:** The memoized implementation[25] of the Fibonacci sequence using the `MemoTrie` package.

The problem with `MemoTrie` is to determine if an input has already been computed, the entire input needs to be traversed through. This works well when the input of a function is a small datatype, but can become problematic when given a large recursive data structure. For example, when we want to compute the summation over a tree, the `lookup` can take up a lot of processing time.

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

sumTree :: BinTree -> Int
sumTree (Leaf x)     = x
sumTree (Node l x r) = x + sumTree l + sumTree r
```

**Figure 1.3:** Computing the summation of all the numbers in the Tree.

To improve the comparison of two recursive data structure, we introduce the use of hash functions. Using hash functions to generate *hash values/digests*, the comparison of two data structures can be performed in constant time. The hash function used can be found in Chapter 2.

To store the digests, we label every node in the tree with its corresponding digest. The digest represents the internal structure of itself and its children. This new data structure is called a *Hash Tree* or *Merkle Tree*[16]. Using the digests within the data structure, the comparison can easily be performed by just comparing both digests for equality.

```haskell
data BinTreeH = Leaf Digest Int
              | Node Digest BinTreeH Int BinTreeH

merkle :: BinTree -> BinTreeH
merkle (Leaf x)     = LeafH (hash ["Leaf", show x]) x
merkle (Node l x r) = NodeH d l' x r'
  where
    d = hash ["Node", show x, getDigest l', getDigest r']
    l' = merkle l
    r' = merkle r
```

**Figure 1.4:** Converting the Tree into a Merkle Tree.

However, what if we want to update a small part of the merkle tree? Then the entire merkle tree gets rehashed, while only a small part is updated. To improve the performance, only the digest of a node has to change if the node itself changes or one of its children changes. To efficiently perform this in Haskell we use a technique named *Zipper*[12]. The inner workings of the Zipper is explained in Section 2.2.

Unfortunately, with this implementation it only works for a single datatype. When we want to support a different datatype the functionality needs to be copied and reimplemented for that specific datatype. This can become quite cumbersome and error-prone for developers. To support a large class of datatypes for this functionality we introduce *Datatype-Generic programming*. The detailed explanation of what datatype-generic programming is, can be found in Chapter 3.

Datatype-generic programming is a technique to exploit the structure of datatypes to define functions by induction over the type structure. To represent datatypes in a generic representation, we use pattern functors. Then using datatype-generic programming, we define generic functionality

for: computing the digests of the data structure, storing the digests inside the data structure, a generic zipper, and, functionality for computing the result and cached results of a given function. The generic implementation of the incremental algorithm is explained in Chapter 4.

However, this does mean that the given function which computes a result needs to use the pattern functors. The pattern functors are quite verbose and the extension cannot be a drop-in replacement for existing functionality. To make it easier for developers to use, we introduce *Pattern synonyms*[21]. Pattern synonyms add an abstraction over patterns, which can be used to simplify the case expressions used in the given function, making the functionality almost a drop-in extension (we only need to add an underscore to the data constructors). The implementation of the pattern-synonyms can be found in Section 4.4.

Finally, to prevent the cache from growing too large for the available amount of memory, we provide multiple policies, so that the developer can choose the best policy for their use-case. The policy can be focused on recency, frequency, computational cost or a combination of the previously mentioned metrics. The suggested cache policies can be found in Section 4.3.

### Example

```
-- The generic implementation for sumTree
sumTree :: PF (BinTree) -> Int
sumTree (Leaf_ x)     = x
sumTree (Node_ l x r) = x + l + r

-- Create a BinTree
> let exampleTree = Node (Node (Leaf 8) 7 (Leaf 1)) 3 (Node (Leaf 5) 4 (Leaf 2))
-- Add digests to the BinTree
> let merkleTree = merkle exampleTree

-- Initial computation
> let (y, m) = cataMerkle sumTree (merkleTree)
    (18, { "6dd": 18, "5df": 15, "fa0": 8, "8d0": 1, "f3b": 9, "84b": 5
         , "1ad": 2 })

-- Update BinTree using the Zipper
> let merkleTree' = update (const (merkle (Leaf 6))) [Bttm] merkleTree

-- Incremental compute the result using
-- the cached results from the previous computation
> cataMerkleMap sumTree m (merkleTree')
    (16, { "6dd": 18, "5df": 15, "fa0": 8, "bbd": 16, "91c": 13, "3af": 6
         , "8d0": 1,  "f3b": 9,  "84b": 5, "1ad": 2 })
```

**Figure 1.5:** The process of using the incremental computation functionality

## 1.1 Contributions

In summary, the main contributions of the Thesis are the following:

- We define an algorithm for incremental computation over recursive data structures. The incremental algorithm uses digests for comparing whether data structures are equal in constant time and a Zipper to efficiently update the recursive data structure without rehashing the entire data structure.
- We use datatype-generic programming to write a generic version of the incremental algorithm, to support a large class of datatypes, namely *regular datatypes*.
- We use pattern synonyms, to make the developer experience the same as implementing a non-incremental algorithm.
- We define cache addition policies and cache replacement policies to optimize the performance/memory usage for different use-cases.

# 2

# Specific Implementation

The comparison of two values for equality in Haskell is performed using the `Eq` typeclass, which implements the equality operator `(==) :: a -> a -> Bool`. An example implementation of the `Eq` typeclass for the `Tree` datatype would be:

```
instance Eq a => Eq (Tree a) where
  Leaf x1       == Leaf x2       = x1 == x2
  Node l1 x1 r1 == Node l2 x2 r2 = x1 == x2 && l1 == l2 && r1 == r2
  _             == _             = False
```

The problem with using this implementation of the `Eq` typeclass for memoization is that for every comparison of the `Tree` datatype the equality is computed. This is inefficient because the equality implementation has to traverse the complete `Tree` data structure to know if the `Tree`'s are equal.

To efficiently compare the `Tree` datatypes, we need to represent the structure in a manner which does not lead to traversing to the complete `Tree` data structure. This can be accomplished using a *hash* function. A hash function is a process of transforming a data structure into an arbitrary fixed-size value, where the same input always generates the same output.

One thing to be cautious when using digests are *hash collisions*. Hash collisions happen when two different inputs have the same resulting hash. This is because a hash function has a limited amount of bits to represent every possible combination of data. To calculate the chance of a hash collision occurring we use the formula $p = \epsilon^{\frac{-k(k-1)}{2N}}$ from *Hash Collision Probabilities*[23]. So, given a common hash function CRC-32[20], which has a digest size of 32bits, to get a 50% chance of a hash collision occurring in a collection, it needs $0.5 = \epsilon^{\frac{-k(k-1)}{2\times 2^{32}}} \rightarrow k = 77163$ hash values.

| Digest size (in bits) | Collection size |
|---|---|
| 32 | 77163 |
| 64 | $5.06 \times 10^9$ |
| 128 | $2.17 \times 10^{19}$ |

**Table 2.1:** The collection size needed for a 50% chance of getting a hash collision

The hash function ultimately chosen in this paper is the *CityHash*[24]. CityHash is a non-cryptographic hash function which works well for hash tables[22]. It supports two different digest

sizes: 64- and 128-bit. The digest size used in this paper will be the 64-bit variant. This digest size is chosen, because we think it gives a good balance between performance and the probability of getting a hash collision.

```
class Hashable a where
  hash :: a -> Digest

instance Show a => Hashable a where
  hash = cityHash64 . show

instance Hashable a => Hashable (Tree a) where
  hash (Leaf x)     = concatHash [hash "Leaf", hash x]
  hash (Node l x r) = concatHash [hash "Node", hash x, hash l, hash r]
```

To keep track of the results of the computation, we store the results in a `HashMap`[27]. A `HashMap` is an implementation of mapping a *hashable* key to a value. The implementation of the HashMap is based on *hash array mapped tries*[3]. The average-case complexity for the `lookup` and `insert` are $\mathcal{O}(\log n)$, however in practice these operations are in constant time. Especially, because the keys are digests, meaning that the key size is constant.

```
sumTreeInc :: Tree Int -> (Int, HashMap Digest Int)
sumTreeInc l@(Leaf x)     = (x, insert (hash l) x empty)
sumTreeInc n@(Node l x r) = (y, insert (hash n) y (ml <> mr))
  where
    y = x + xl + xr
    (xl, ml) = sumTreeInc l
    (xr, mr) = sumTreeInc r
```

Then after the first computation over the entire `Tree`, we can recompute the `Tree` using the previously created `HashMap`. Thus, when we recompute the `Tree`, we first look in the `HashMap` if the computation has already been performed then return the result. Otherwise, compute the result and store it in the `HashMap`.

```
sumTreeIncMap :: HashMap Digest Int -> Tree Int -> (Int, HashMap Digest Int)
sumTreeIncMap m l@(Leaf x) = case lookup (hash l) m of
  Just x  -> (x, m)
  Nothing -> (x, insert (hash l) x empty)
sumTreeIncMap m n@(Node l x r) = case lookup (hash n) m of
  Just x  -> (x, m)
  Nothing -> (y, insert (hash n) y (ml <> mr))
    where
      y = x + xl + xr
      (xl, ml) = sumTreeIncMap m l
      (xr, mr) = sumTreeIncMap m r
```

Generating a hash for every computation over the data structure is time-consuming and unnecessary, because most of the `Tree` data structure stays the same. The work of Miraldo and Swierstra[18] inspired the use of the Merkle Tree. A Merkle Tree is a data structure which integrates the digests within the data structure.

## 2.1 Merkle Tree

First we introduce a new datatype `TreeH`, which contains a `Digest` for every constructor in `Tree`. Then to convert the `Tree` datatype into the `TreeH` datatype, the structure of the Tree is hashed and stored into the datatype using the `merkle` function.

```
data TreeH a = LeafH Digest a
             | NodeH Digest (Leaf a) a (Leaf a)

merkle :: Tree Int -> TreeH Int
merkle (Leaf x)     = LeafH (hash ["Leaf", show x]) x
merkle (Node l x r) = NodeH d l' x r'
  where
    d = hash ["Node", show x, getDigest l', getDigest r']
    l' = merkle l
    r' = merkle r
```

The precomputed digests can then be used to easily create a `HashMap`, without computing the digests every time the `sumTreeIncH` function is called.

```
sumTreeIncH :: TreeH Int -> (Int, HashMap Digest Int)
sumTreeIncH (LeafH h x)     = (x, insert h x empty)
sumTreeIncH (NodeH h l x r) = (y, insert h y (ml <> mr))
  where
    y = x + xl + xr
    (xl, ml) = sumTreeInc l
    (xr, mr) = sumTreeInc r
```

The problem with this implementation is, that when the `Tree` datatype is updated, the entire `Tree` needs to be converted into a `TreeH`, which is linear in time. This can be done more efficiently, by only updating the digests which are impacted by the changes. Which means that only the digests of the change and the parents need to be updated.

The first intuition to fixing this would be using a pointer to the value that needs to be changed. But because Haskell is a functional programming language, there are no pointers. Luckily, there is a technique which can be used to efficiently update the data structure, namely the Zipper[12].

## 2.2 Zipper

The Zipper is a technique for keeping track of how the data structure is being traversed through. The Zipper was first described by Huet[12] and is a solution for efficiently updating pure recursive data structures in a purely functional programming language (e.g., Haskell). This is accomplished by keeping track of the downward current subtree and the upward path, also known as the *location*.

To keep track of the upward path, we need to store the path we traverse to the current subtree. The traversed path is stored in the `Cxt` datatype. The `Cxt` datatype represents three options the path could be at: the `Top`, the path has traversed to the left (`L`), or the path has traversed to the right (`R`).

```
data Cxt a = Top
           | L (Cxt a) (Tree a) a
           | R (Cxt a) (Tree a) a

type Loc a = (Tree a, Cxt a)

enter :: Tree a -> Loc a
enter t = (t, Top)
```

Using the `Loc`, we can define multiple functions on how to traverse through the `Tree`. Then, when we get to the desired location in the `Tree`, we can call the `modify` function to change the `Tree` at the current location.

Eventually, when every value in the `Tree` has been changed, the entire `Tree` can then be rebuilt using the `Cxt`. By recursively calling the `up` function until the top is reached, the current subtree gets rebuilt. And when the top is reached, the entire tree is then returned.

```
left :: Loc a -> Loc a
left (Node l x r, c) = (l, L c r x)

right :: Loc a -> Loc a
right (Node l x r, c) = (r, R c l x)

up :: Loc a -> Loc a
up (t, L c r x) = (Node t x r, c)
up (t, R c l x) = (Node l x t, c)

modify :: (Tree a -> Tree a) -> Loc a -> Loc a
modify f (t, c) = (f t, c)

leave :: Loc a -> a
leave (t, Top) = t
leave l        = top (up l)
```

```
> leave $ modify (const (Leaf 4)) $ left $ enter (Node (Leaf 1) 2 (Leaf 3))
    (Node (Leaf 4) 2 (Leaf 3))
```

### 2.2.1 Zipper Merkle Tree

The implementation of the Zipper for the `TreeH` datatype is the same as for the `Tree` datatype. However, the `TreeH` also contains the hash of the current and underlying data structure. Therefore, when a value is modified in the `TreeH`, all the parent digests of the modified value needs to be updated.

The `updateLoc` function modifies the value at the current location, then checks if the location has any parents. If the location has any parents, go up to that parent, update the digest of that parent and recursively update the parents digests until we are at the top of the data structure. Otherwise, return the modified locations, because all the other digests are not affected by the change.

```
updateLoc :: (TreeH a -> TreeH a) -> Loc a -> Loc a
updateLoc f l = if top l' then l' else updateParents (up l')
  where
    l' = modify f l
    updateParents :: Loc a -> Loc a
    updateParents (Loc x Top) = Loc (updateHash x) Top
    updateParents (Loc x cs)  = updateParents $ up (Loc (updateHash x) cs)
```

Then, the `update` function can be defined using the `updateLoc` function, by first traversing through the data structure with the given directions. Then modifying the location using the `updateLoc` function and then leave the location and the function results in the updated data structure.

```
update :: (TreeH a -> TreeH a) -> [Loc a -> Loc a] -> TreeH a -> TreeH a
update f dirs t = leave $ updateLoc f l'
  where
    l' = applyDirs dirs (enter t)
```

# 3

# Datatype-Generic Programming

The implementation in Chapter 2 is an efficient implementation for incrementally computing the summation over a `Tree` datatype. However, when we want to implement this functionality for a different datatype, a lot of code needs to be copied while the process remains the same. This results in poor maintainability, is error-prone and is in general boring work.

An example of reducing manual implementations for datatypes is the *deriving* mechanism in Haskell. The built-in classes of Haskell, such as `Show, Ord, Read`, can be derived for a large class of datatypes. However, *deriving* is not supported for recursive datatypes. Therefore, we use *Datatype-Generic Programming*[9] to define functionality for a large class of datatypes.

In this chapter, we introduce Datatype-Generic Programming, also known as *generic programming* in Haskell, as a technique that uses the structure of a datatype to define functions for a large class of datatypes. This prevents the need to write the previously defined functionality for every datatype.

## 3.1 Introduction

There are multiple generic programming libraries, however to demonstrate the workings of generic programming we will be using a single library as inspiration, named `regular`[15]. Here the generic representation of a datatype is called a *pattern functor*. A pattern functor is a stripped-down version of a datatype, by only containing the constructor but not the recursive structure. The recursive structure is done explicitly by using a fixed-point operator.

First, the pattern functor defined in `regular` are 5 primitive type constructors and 2 meta information constructors. The primitive type constructors describe the datatypes. The meta information constructors only contain information (e.g., constructor name) but not any structural information.

```
data U r          = U                  -- Empty constructor
data I r          = I r                -- Recursive call
data K a r        = K a                -- Constant
data (f :+: g) r = L (f r) | R (g r) -- Sums (Choice)
data (f :*: g) r = (f r) :*: (g r)    -- Products (Combine)
```

The conversion from datatypes into pattern functors is done by the `Regular` type class. The `Regular` type class has two functions. The `from` function converts the datatype into a pattern functor and the `to` function converts the pattern functor back into a datatype. In `regular`, the pattern functor is represented by a type family. Then using the `Regular` conversion to a pattern functor, we can write the `Tree` datatype from Chapter 2 as:

```
type family PF a :: * -> *

class Regular a where
  from :: a -> PF a a
  to   :: PF a a -> a

type instance PF (Tree a) = K a                 -- Leaf
                          :+: (I :*: K a :*: I) -- Node

class Regular (Tree a) where
  from (Leaf x)     = L (K x)
  from (Node l x r) = R (I l :*: K x :*: I r)

  to (L (K x))                 = Leaf x
  to (R (I l :*: K x :*: I r)) = Node l x r
```

To demonstrate the workings of generic programming, we are going to implement a simple generic function which determines the length of an arbitrary datatype. First, we define the length function within a type class. The type class is used, to define how to compute the length for every primitive type constructor `f`.

```
class GLength f where
  glength :: (a -> Int) -> f a -> Int
```

Writing instances for the empty constructor `U` and the constants `K` is simple because both primitive type constructors return zero. The `U` returns zero, because it does not contain any children. The `K` returns zero, because we do not count constants for the length.

```
instance GLength U where
  glength _ _ = 0

instance GLength (K a) where
  glength _ _ = 0
```

The instances for sums and products are quite similar. The sums recurses into the specified choice. The product recurses in both constructors and combines them.

```
instance (GLength f, GLength g) => GLength (f :+: g) where
  glength f (L x) = glength f x
  glength f (R x) = glength f x

instance (GLength f, GLength g) => GLength (f :*: g) where
  glength f (x :*: y) = glength f x + glength f y
```

The instance for the recursive call `I` needs an additional argument. Because, we do not know the type of `x`, so, an additional function (`f :: a -> Int`) needs to be given which converts `x` into the length for that type.

```
instance GLength I where
  glength f (I x) = f x
```

Then using the `GLength` instances for all primitive type constructors, a function can be defined using the generic length function. By first, converting the datatype into a generic representation, then calling `glength` with the function `length` recursively, and for every recursive call increase the length by one.

```
length :: (Regular a, GLength (PF a)) => a -> Int
length = 1 + glength length (from x)
```

```
> length [1, 2, 3]
    3
> length (Node (Leaf 1) 2 (Leaf 3))
    3
> length {"1": 1, "2": 2, "3": 3}
    3
```

## 3.2 Explicit recursion

The previous implementation of the `length` function is implemented for a shallow representation. A shallow representation means that the recursion of the datatype is not explicitly marked. Therefore, we can only convert one layer of the value into a generic representation using the `from` function.

Alternatively, by marking the recursion of the datatype explicitly, also called the deep representation, the entire value can be converted into a generic representation in one go. To mark the recursion, a fixed-point operator (`Fix`) is introduced. Then, using the fixed-point operator we can define a `from` function that given the pattern functors have an instance of `Functor`[1], return a generic representation of the entire value.

[1] The `Functor` instances for the pattern functors can be found in Appendix A.1

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

deepFrom :: (Regular a, Functor (PF a)) => a -> Fix (PF a)
deepFrom = In . fmap deepFrom . from
```

Subsequently, we can define a `cata` function which can use the explicitly marked recursion by applying a function at every level of the recursion. Then using the `cata` function we can define the same `length` function as in the previous section, but just in a single line. However, this deep representation does come at the cost that the implementation is less efficient than the shallow representation.

```haskell
cata :: Functor f => (f a -> a) -> Fix f -> a
cata = f . fmap (cata f) . unFix

length' :: (Regular a, GLength (PF a), Functor (PF a), Foldable (PF a))
        => a -> Int
length' = cata ((1+) . sum) . deepFrom
```

## 3.3 Sums of Products

A different way of describing datatypes in a generic representation, besides pattern functors, are *Sums of Products*[28] (SOP). SOP is a generic representation with additional constraints compared to pattern functors, which more faithfully reflects the Haskell datatypes. Each datatype is a single n-ary sum, where each component of the sum is a single n-ary product. In SOP, the generic representation is first described as a *code* of kind `[[*]]`. The outer list describes an n-ary sum, representing the choice between constructors and each inner list an n-ary products, representing the constructor arguments. The code is not the same as the generic representation, the code is the structure which the generic representation has to satisfy. Therefore, a mapping is needed which converts the code of kind `[[*]]` into a generic representation of kind `*`. The code is defined using a tick mark `` ` `` and is used to lift the list from a data-level to a type-level.

```haskell
Code (Tree a) = `[`[a], `[Tree a, a, Tree a]]
```

The usage of SOP has a positive effect on expressing generic functions easily or at all. Additionally, the SOP completely divides the structural representation from the metadata. As a result, you do not have to deal with metadata while writing generic functions.

However, SOP uses type-level lists to put additional constraints onto the generic representation, while pattern functors does not. This makes extending the generic functionality more complex for SOP than pattern functors, because besides data-level programming, also the type-level programming has to be correct.

## 3.4 Mutually recursive datatypes

A large class of datatypes is supported by the previous section, namely *regular* datatypes. Regular datatypes are datatypes in which the recursion only goes into the same datatype. However, if we want to support the abstract syntax tree (AST) of many programming languages, we need to support datatypes which can recurse over different datatypes, namely mutually recursive datatypes.

```
data Tree a = Empty
            | Node (a, Forest a)

data Forest a = Nil
              | Cons (Tree a) (Forest a)
```

To support mutually recursive datatypes, we need to keep track of which recursive position points to which datatype. This is accomplished by using an *indexed fixed-point*[29]. The indexed fixed-point works by creating a type family $\varphi$ with $n$ different types, where the types inside the family represent the indices for different kinds ($*_{\varphi}$). Using the limited set of kinds we can determine the type for the recursive positions. Thus, supporting mutually recursive datatypes is possible, but it adds more complexity to the implementation.

# 4

# Generic Implementation

## 4.1 Regular

Ultimately, the generic programming library chosen to implement the incremental computation is the `regular` library. The `regular` library is chosen, because the library is the easiest to use, while still supporting enough datatypes to have meaningful results. However, the mutually recursive datatypes are not supported, and it does not use SOP which can cause more bugs to occur, because pattern functors do not represent Haskell datatypes as correctly as SOP.

The first step of writing the generic incremental computation was computing the Merkle Tree. In other terms, we need to store the hash of the data structure inside the data structure. We accomplish this by defining a new type `Merkle` which is a fixed-point over the data structure where each of the recursive positions contains a digest of type (`K Digest`).

```
type Merkle f = Fix (f :*: K Digest)
```

But, before the hash can be stored inside the data structure, the hash needs to be computed from the data structure. For this we need to know how to hash the primitive type constructors. We introduce a typeclass named `Hashable` which defines a function `hash`, which converts the `f` datatype into a `Digest`.

```
class Hashable f where
  hash :: f (Merkle g) -> Digest
```

The Hashable instance of `U` is simple. The `hash` function is used to convert the constructor name `U` into a `Digest`. The `K` also uses the `hash` function to convert the constructor name into a `Digest`, but it also calls `hash` on the constant value of `K`. Therefore, the type of the value of `K` needs an instance for `Show`. Then both digests are combined into a single digest.

```
instance Hashable U where
  hash _ = hash "U"

instance (Show a) => Hashable (K a) where
  hash (K x) = digestConcat [hash "K", hash x]
```

The instances for sums and products are quite similar as the instance for the `K` datatype. However, the value inside the constructor are recursively called.

```
instance (Hashable f, Hashable g) => Hashable (f :+: g) where
  hash (L x) = digestConcat [hash "L", hash x]
  hash (R x) = digestConcat [hash "R", hash x]

instance (Hashable f, Hashable g) => Hashable (f :*: g) where
  hash (x :*: y) = digestConcat [hash "P", hash x, hash y]
```

The `I` instance is different from the previous instances, because the recursive position is already converted into a Merkle Tree. Thus, we need to get the computed hash from the recursive position, digest the datatype name and combine the digests.

```
instance Hashable I where
  hash (I x) = digestConcat [digest "I", getDigest x]
    where
      getDigest :: Fix (f :*: K Digest) -> Digest
      getDigest (In (_ :*: K h)) = h
```

The `hash` implementation can then be used to define a function `merkleG` which converts from a shallow generic representation, to a generic representation where one layer of recursive positions contains a hash value.

Subsequently, we can define a function `merkle` which converts the entire datatype, into a pattern functor where every recursive position contains a hash value. We can define `merkle` using the same implementation as in Section 3.2, but we add a step where after all the children are recursively called, the `merkleG` function is applied.

```
merkleG :: Hashable f => f (Merkle g) -> (f :*: K Digest) (Merkle g)
merkleG f = f :*: K (hash f)

merkle :: (Regular a, Hashable (PF a), Functor (PF a))
       => a -> Merkle (PF a)
merkle = In . merkleG . fmap merkle . from
```

The `Merkle` representation can then be used to define a function `cataMerkleState` which given a function `alg :: (f a -> a)` which converts the generic representation `f a` into a value of type `a` and the `Merkle f` data structure, and returns a `State` of `(HashMap Digest a) a`. The `cataMerkleState` function starts with retrieving the `State`, which keeps track of the cached results and stores them into a `HashMap Digest a`. Then, given the hash value of the recursive position, we look into the HashMap if the value has been computed. If the value has been computed, then return the value. Otherwise, recursively compute all the children, apply the given function `alg`, insert the new value into the `HashMap` and return the computed value.

```
cataMerkleState :: (Functor f, Traversable f)
                => (f a -> a) -> Merkle f -> State (HashMap Digest a) a
cataMerkleState alg (In (x :*: K h))
  = do m <- get
       case lookup h m of
         Just a  -> return a
         Nothing -> do y <- mapM (cataMerkleState alg) x
                       let r = alg y
                       modify (insert h r) >> return r
```

The `cataMerkleState` function can be used, but to execute the function, first the function needs a `HashMap Digest a`. To simplify the use of `cataMerkleState`, a function `cataMerkle` is defined, which executes the `cataMerkleState` with an empty `HashMap` and returns the final computed result and the final state as the result.

```
cataMerkle :: (Functor f, Traversable f)
           => (f a -> a) -> Merkle f -> (a, HashMap Digest a)
cataMerkle alg t = runState (cataMerkleState alg t) empty
```

Finally, all the necessary functionality is defined to write a function over the generic representation and automatically generate all the cached results and the final result. The example below computes the sum over the generic representation of `Tree` by adding all the values of the leaf and nodes.

```
cataSum :: Merkle (PF (Tree Int)) -> (Int, HashMap Digest Int)
cataSum = cataMerkle
  (\case
    L (C (K x))                 -> x
    R (C (I l :*: K x :*: I r)) -> l + x + r
  )

> cataSum $ merkle $ Node (Leaf 1) 2 (Leaf 3)
    (6, {"931090e5": 1, "7d1ef1c9": 3, "ba811ed5": 6})
```

## 4.2 Generic Zipper

For the implementation of a generic zipper, we need to define (A) a datatype which keeps track of the location inside the data structure, (B) a context which keeps track of the locations which have been traversed through (C) and functions which facilitate traversing through the data structure. The implementation of the general zipper is based on the paper "Generic representations of tree transformations" by Bransen and Magalhaes[5].

The context (`Ctx`) is implemented using a type family[1][13]. Type families can be defined in two manners, standalone or associated with a type class. For the explanation we use the standalone definition because the code is less clumped, making it easier to explain. However, the actual implementation uses type synonym families, which is makes it clearer how the type should be used and has better error messages.

The standalone definition uses a `data family`. Then, we want to write for every representation type an instance on how to represent the representation type in the context.

```
data family Ctx (f :: * -> *) :: * -> *
```

The `K` and `U` representation-types do not have a datatype, because these representation-types cannot be traversed through.

```
data instance Ctx (K a) r
data instance Ctx U     r
```

The sum representation-type does get traversed, but only has one choice by either traversing the left `CL` or the right `CR` side.

```
data instance Ctx (f :+: g) r = CL (Ctx f r) | CR (Ctx g r)
```

The product representation-type does have a choice between two traversals, either traverse through the left side and store the right side or traverse through the right side and store the left side.

```
data instance Ctx (f :*: g) r = C1 (Ctx f r) (g r) | C2 (f r) (Ctx g r)
```

The recursive representation-type does not recursively go into a new `Ctx`, but into the recursive type `r`, thus we only need to define datatype which indicates that it is a recursive position.

```
data instance Ctx I r = CId
```

The `Zipper` type class can now be defined using the `Ctx`. There are 6 primary functions, which we need to build navigating functions. The `cmap` function works like the `fmap`, but over contexts. The `fill` function fills the hole in a context with a given value, which is used to reconstruct the data structure. The final 4 functions (`first`, `last`, `next` and `prev`) are the *primary* navigation operations used to build *interface* functions such as `left`, `right`, `up`, `down`, etc.

[1] "*Type families are to vanilla data types what type class methods are to regular functions.*"[10]

```
class Functor f => Zipper f where
  cmap        :: (a -> b) -> Ctx f a -> Ctx f b
  fill        :: Ctx f a -> a -> f a
  first, last :: f a -> Maybe (a, Ctx f a)
  next, prev  :: Ctx f a -> a -> Maybe (a, Ctx f a)
```

Finally, we can define the location `Loc` using the `Ctx` and the `Zipper`. The location takes a datatype with the constraints that it has an instance for `Regular` and a pattern functor which works with the `Zipper` type class, a list of contexts and returns a location datatype `Loc`.

```
data Loc :: * -> * where
  Loc :: (Regular a, Zipper (PF a)) => a -> [Ctx (PF a) a] -> Loc a
```

To use the `Merkle` type, we need to fulfill the previous constraints. Thus, we need to define a type instance for the pattern functor and an instance for the `Regular` typeclass. The pattern functor type instance is the same definition as the `Merkle` type but without the fixed-point. The `Regular` instance for the `Merkle` type is folding/unfolding the fixed-point. Moreover, the `Merkle` type also needs to update the digests of its parents when a value is changed. This is accomplished in the same manner as in Section 2.2.1.

```
type instance PF (Merkle f) = f :*: K Digest
instance Regular (Merkle f) where
  from = out
  to   = In
```

## 4.3 Cache Management

In the optimal case, each cached value can be stored. Unfortunately, we are limited by the amount of memory there is available on the machine. Therefore, the amount of cached results needs to be limited. There are two suggestions of limiting the amount of cached results with *Cache Addition Policies* and *Cache Replacement Policies*. Cache addition policies are policies which indicate if a cached value can be added to the cache. And the cache replacement policies are policies which remove cached results from the cache based on metrics (e.g., recency, frequency, computational cost, etc.).

### 4.3.1 Cache Addition Policies

An example of a cache addition policy would be to determine the recursion depth a data structure has. So, given a node in a tree, how many times we have to go into recursion before reaching the deepest leaf is the recursion depth. Using that information we can define a filter for the cache, where only elements with a recursion depth of $> i$ can be added to the cache. This filters out a large amount of cached results and could potentially lead to a speed-up, because the lookup in the HashMap takes longer when there are more elements.

The recursion depth can be determined in the same pass as when the digests are calculated. To accomplish this we introduce an annotated fix-point. The annotated fix-point contains the fix-point as explained in Section 3.2 and an annotation for which we store the information about the digest and the recursion depth. The final `Merkle` type is the annotated fix-point with the annotation containing the digest and the recursion depth. The implementation of the determining the recursion depth is in Appendix B.1.

```
data AFix f a = AFix { unAFix :: f (AFix f a), getAnnotation :: a }

data MemoInfo = MemoInfo { getDigest :: Digest, getDepth :: Int }

type Merkle f = AFix f MemoInfo
```

Then to compute the digest and the recursion depth over a generic datatype, the `merkle` function has to be expanded. An additional type constraints `HasDepth (PF a)` is added for the new typeclass which computes the recursion depth. Then the digest and depth is computed over the pattern functor and the final annotated fix-point with the digest and recursion depth is returned.

```
merkle :: (Regular a, Hashable (PF a), HasDepth (PF a), Functor (PF a))
       => a -> Merkle (PF a)
merkle x = AFix py (MemoInfo d h)
  where
    py = merkle <$> from x
    d  = hash py
    h  = depth py

-- Updated cataMerkleState to support cache addition policy for recursion depth
cataMerkleState :: (Functor f, Traversable f)
                => (f a -> a) -> AFix f MemoInfo
                -> State (HashMap Digest (a, CacheInfo)) a
cataMerkleState alg (AFix x (MemoInfo di de))
  = do m <- get
       case lookup di m of
         Just a -> return a
         Nothing -> do y <- mapM (cataMerkleState alg) x
                       let r = alg y
                       if minDepth <= de
                         then  modify (insert d r) >> return r
                       else return r
```

However, this does not prevent the memory from filling-up. A more drastic measure is needed to keep the memory from filling up and that is to remove cache results. For this another policy needs to be defined and that are the cache replacement policies.

### 4.3.2 Cache Replacement Policies

The cache replacement policy can be based on a single metric of a combination of multiple metrics. Unfortunately, we cannot determine the overall best cache replacement policy, because the best cache replacement policy is application specific as stated in "Selective memoization"[2]. As a result, this paper will only describe possible policies which can be used by developers, but not show any results.

Fortunately, it is quite simple to implement the cache replacement policies. At the end of the `cataMerkle` computation, a filter is passed over the cache results. If the cached value is true with the current cache replacement policy, it gets discarded. Otherwise, the cached value stays in the cache.

```
data CacheInfo = CacheInfo { getFrequency :: Int, getRecency :: Int }

  -- An example for a replacement policy
replacementPolicy :: CacheInfo -> Bool
replacementPolicy c = getFrequency c <= 1

applyPolicy :: HashMap Digest (a, CacheInfo) -> HashMap Digest (a, CacheInfo)
applyPolicy = filter (replacementPolicy . snd)

cataMerkle :: (Functor f, Traversable f)
           => (f a -> a) -> AFix f MemoInfo -> (a, HashMap Digest (a, CacheInfo))
cataMerkle alg t = (y, applyPolicy ys)
  where
    (y, ys) = runState (cataMerkleState alg t) empty
```

There are a multitude of replacement policies which can be chosen to choose which cached results needs to be replaced. Below, there are some suggestions of what type of replacement policies can be chosen.

- Random replacement: remove random elements from the cache.
- Recency-based policies: remove elements based on when the element was added.
- Frequency-based policies: remove elements based on the amount of lookups.
- Computational cost policies: remove elements based on the amount of time it takes to compute the value.
- Combination of policies mentioned above. For example, combine the frequency based policy with the computation cost policy.

[2]"In general the replacement policy must be application-specific, because, for any fixed policy, there are programs whose performance is made worse by that choice."[2]

## 4.4 Pattern Synonyms

The developer experience using `cataMerkle` is difficult, because the developer needs to know the pattern functor of its datatype to define a function and the function definitions are quite verbose. To make the use of `cataMerkle` easier, we introduce *pattern synonyms*[21].

Pattern synonyms add an abstraction over patterns, which allows the user to move additional logic from guards and case expressions into patterns. For example, the pattern functor of the `Tree` datatype can be represented using a `pattern`.

```
pattern Leaf_ :: a -> PF (Tree a) r
pattern Leaf_ x <- L (C (K x)) where
  Leaf_ x = L (C (K x))

pattern Node_ :: r -> a -> r -> PF (Tree a) r
pattern Node_ l x r <- R (C (I l :*: K x :*: I r)) where
  Node_ l x r = R (C (I l :*: K x :*: I r))
```

The previously defined `pattern`s can then be used to define the `cataSum` as the original datatype `Tree`, but the constructor names leading with an additional underscore.

```
cataSum :: Merkle (PF (Tree Int)) -> (Int, HashMap Digest Int)
cataSum = cataMerkle
  (\case
    Leaf_ x     -> x
    Node_ l x r -> l + x + r
  )
```

# 5

# Experiments

## 5.1 Method

We conducted experiments over three type of functions: the `Cata Sum`, `Generic Cata Sum` and `Incremental Cata Sum`. The Cata Sum is the simple function which traverses through the entire tree and sums all the values. The Generic Cata Sum is the initial Incremental Cata Sum, which starts with an empty HashMap. And the Incremental Cata Sum, which already has a HashMap filled with cached results and keeps track over multiple iterations.

```
cataSum :: Tree Int -> Int
cataSum (Leaf x)     = x
cataSum (Node l x r) = x + cataSum l + cataSum r

genericCataSum :: Merkle (PF (Tree Int)) -> (Int, HashMap Digest Int)
genericCataSum = cataMerkle
  (\case
    Leaf_ x     -> x
    Node_ l x r -> l + x + r
  )

incCataSum :: HashMap Digest Int
           -> Merkle (PF (Tree Int)) -> (Int, HashMap Digest Int)
incCataSum = cataMerkleMap
  (\case
    Leaf_ x     -> x
    Node_ l x r -> l + x + r
  )
```

The experiments will be benchmarks executed with the Haskell package `criterion`[19]. Criterion performs the benchmarks multiple times to get an average result. The benchmarks will track two metrics: the *execution time* and the *memory usage*. The execution time will be in seconds and the memory usage will be the max-bytes used. The results gathered for the execution time comes from the criterion package, however the memory usage will not come from the package. This is

because criterion only keeps track of the memory allocation and not usage. Therefore, we measure the memory usage with the GHC profiler[26] and profile every benchmark individually to know its memory usage.

To test how well the three functions perform, we perform three types of updates, multiple times. These benchmarks will be based on the three type of cases: worst, average and best. The worst case updates the lowest left leaf to a new leaf The average case updates a node in the middle with a new leaf And the best case replaces the left child of the root-node with a new leaf.

```
worstCase :: Merkle (PF (Tree Int)) -> Merkle (PF (Tree Int))
worstCase = update (const (Leaf i)) [Bttm]

averageCase :: Merkle (PF (Tree Int)) -> Merkle (PF (Tree Int))
averageCase = update (const (Leaf i)) (replicate n Dwn)
  where
    n = round (logBase 2.0 (fromIntegral n) / 2.0)

bestCase :: Merkle (PF (Tree Int)) -> Merkle (PF (Tree Int))
bestCase = update (const (Leaf i)) [Dwn]
```

## 5.2 Results

The experiments are performed on a laptop with a `Intel Core i7-8750H` with a base clock of `2.2GHz` and a boost clock of `4.1GHz`, with `16GB` of memory. First we explain the results of the three algorithms with three different scenarios, iterating 10 times. Then, we show the results of adding a cache addition policy.

### 5.2.1 Execution Time

Looking at Figure 5.1 the Incremental Cata Sum is faster when the tree contains more than $10^3$ nodes. However, for every benchmark the execution time is better or worse depending on the type of update that is performed. The best case scenario has the biggest difference, then the average case and the closest execution time difference is the worst case. The execution time for Cata Sum and Generic Cata Sum seems to be linear with the amount of nodes and the Incremental Cata Sum is constant/logarithmic. The Cata Sum seems to be a factor faster than the Generic Cata Sum, which makes sense because Generic Cata Sum does the same computation as the Cata Sum. And, additionally computes the generic representation of the data structure, computes the digests for the data structure and stores all the cached result into a `HashMap`. The Incremental Cata Sum is constant/logarithmic, because it only updates the digests of the changed nodes and its parent nodes and recomputes the nodes with changed digests, which is worst case $\mathcal{O}(M \log N)$ where $M$ is the amount of changes and $N$ is the size of the tree. The discrepancy in the range between $10^2 - 10^3$ is probably because the amount of nodes is too low to get stable results.

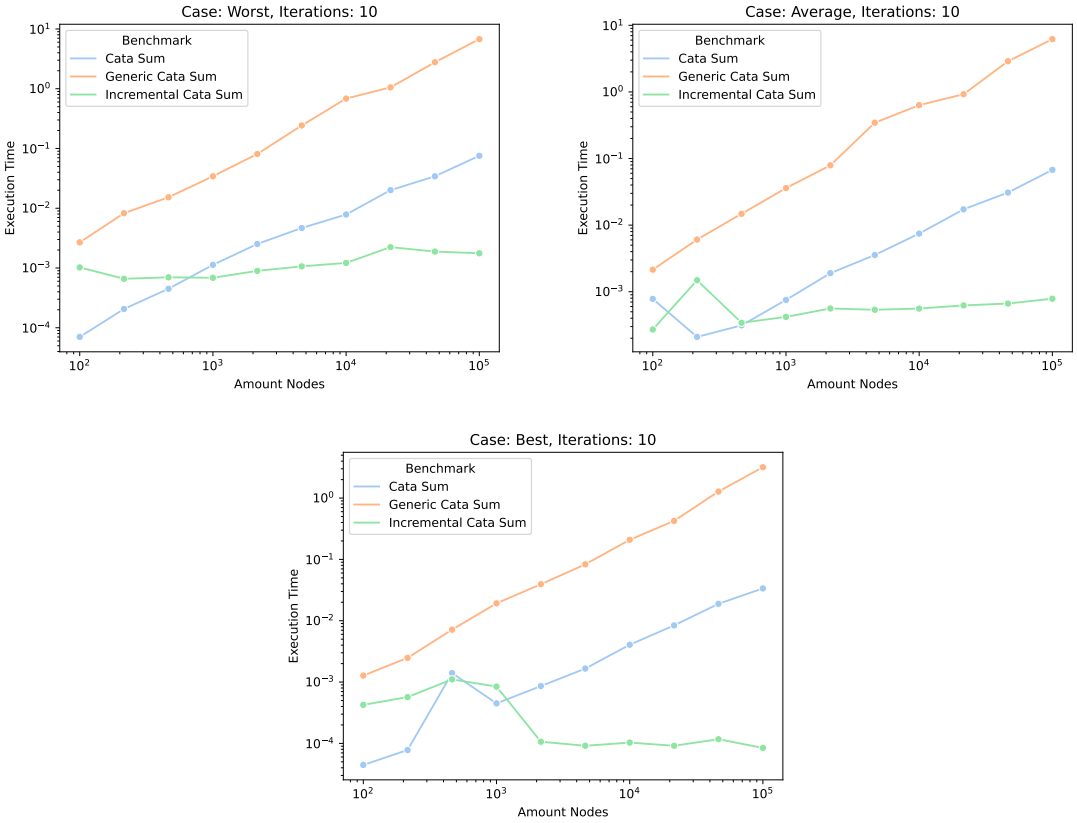


**Figure 5.1:** The execution time over 10 executions for the Worst, Average and Best case.

### 5.2.2 Memory Usage

The Figure 5.2 shows that the memory usage is, for all algorithms, linear with the amount of nodes. The Cata Sum uses the least amount of memory, then the Incremental Cata Sum and the most used bytes is the Generic Cata Sum. We think that the Incremental Cata Sum uses less memory than the Generic Cata Sum, because the Incremental Cata Sum needs to load-in fewer parts of the data structure than the Generic Cata Sum. Also, the same discrepancy occurs in the range between $10^2 - 10^3$ as with the execution time.

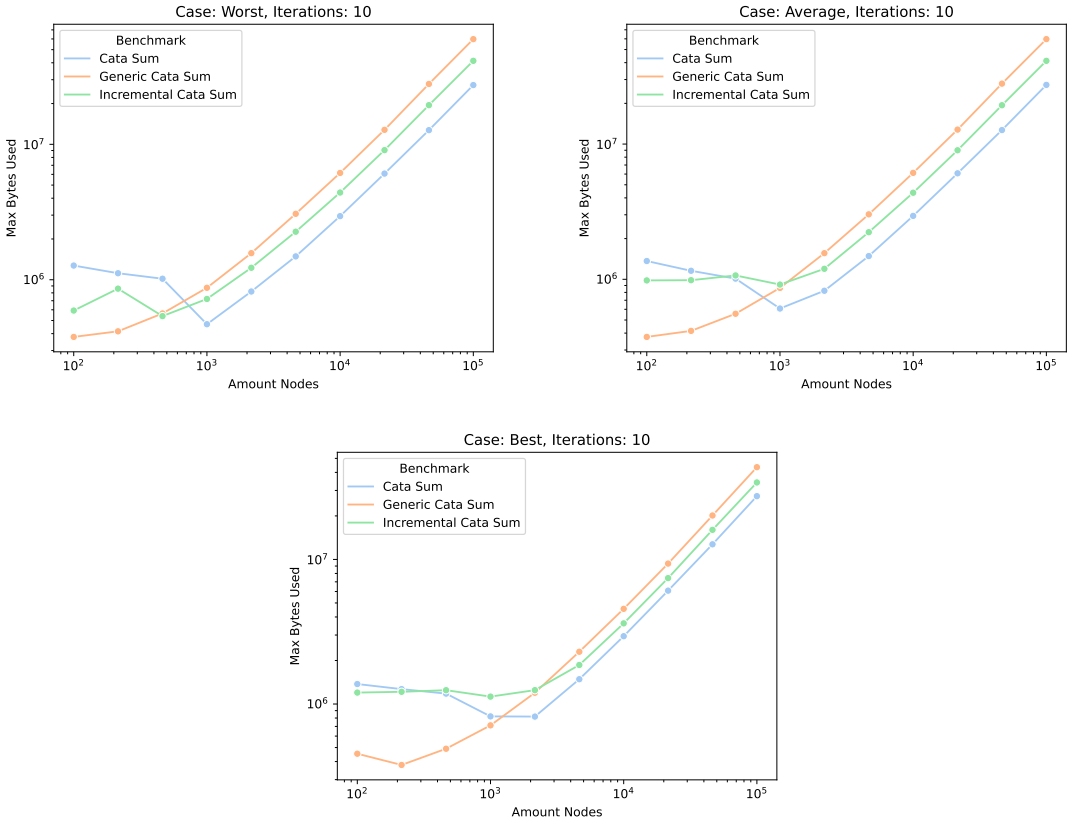


**Figure 5.2:** The max-bytes-used over 10 executions for the Worst, Average and Best case.

### 5.2.3 Comparison Cache Addition Policies

As seen in Figure 5.3 and Figure 5.4, the memory usage is lower than when all the cached results are stored. Even, the execution time did not increase significantly. However, this is probably because the function which is computed (addition) is not computational intensive. This means it is less computational intensive to recompute the value than to store the value and retrieve it when needed. This does not have to be the case for more extensive computations. So, for the best performance this parameter needs to be correctly tweaked.

For example, if we increased the recursion depth limit to 10. The results in Appendix Section C.1 show that the memory usage is even lower than the limit of 5. However, the execution time increases significantly. This is because the computation takes longer to perform than to store and perform a lookup.

## Execution Time

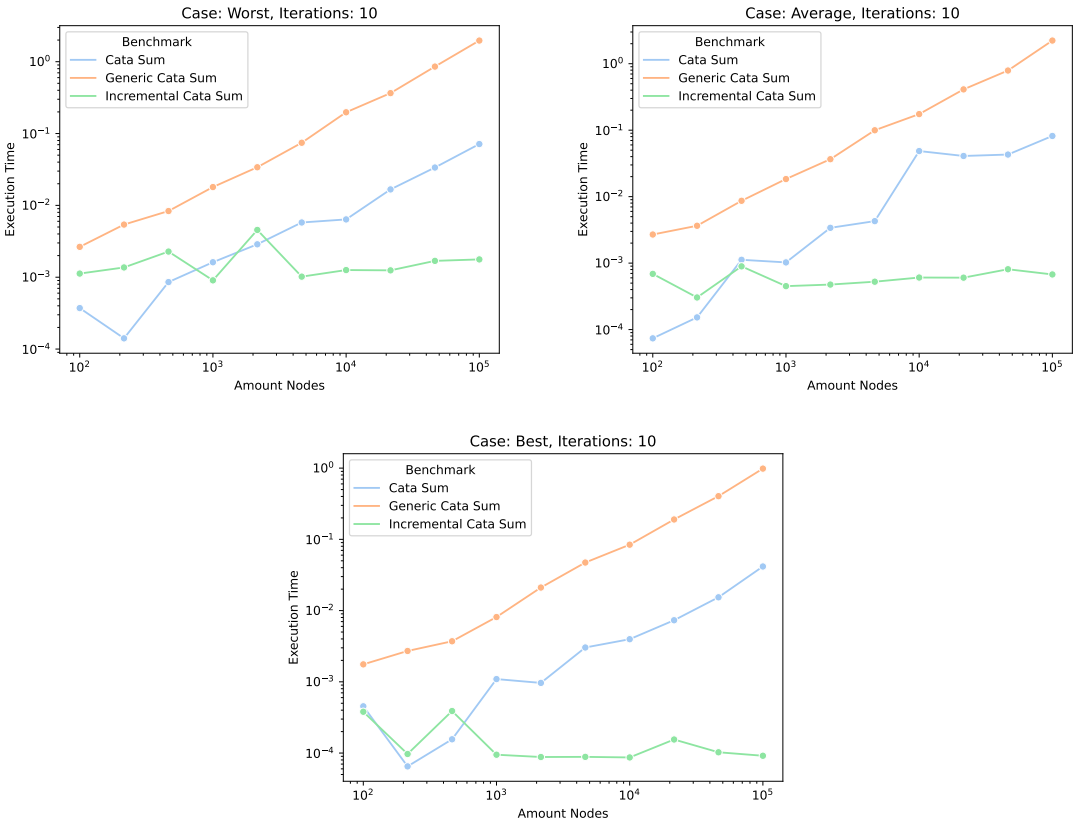


**Figure 5.3:** The execution time over 10 executions for the Worst, Average and Best case, where the minimum recursion depth is 5.

## Memory Usage

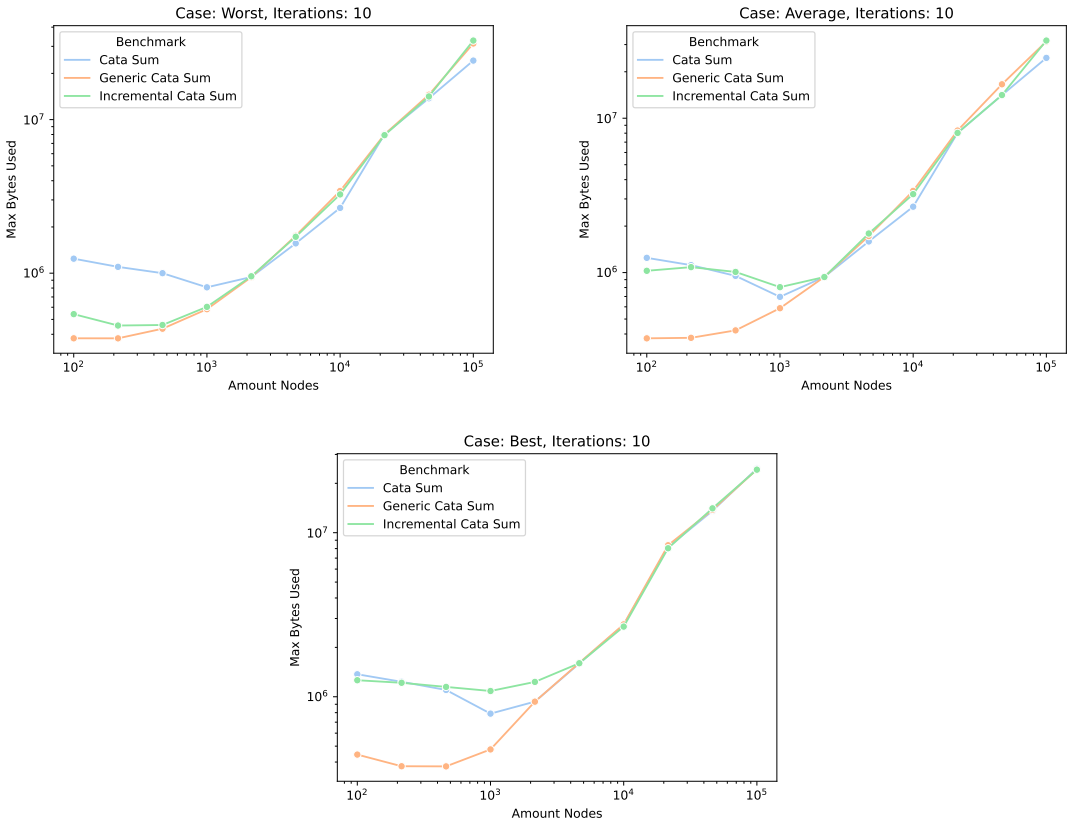


**Figure 5.4:** The max-bytes-used over 10 executions for the Worst, Average and Best case, where the minimum recursion depth is 5.

# 6

# Discussion & Conclusion

## 6.1 Related Work

There are many other studies which implement incremental computation in functional languages[11] [1] [8] [6]. However, each study has a different approach in implementing incremental computation. The most important part of incremental computation is to know if the input has changed and what has changed.

### 6.1.1 Comparison of equality in constant time

To determine the changes between the old and current input can be done by comparing the inputs for equality. This makes it very important that the comparison of the input for equality can be performed efficiently. The problem with most studies is that for every comparison the entire data structure of the input needs to be traversed through (e.g., a tree), which can become every inefficient when folding over a tree [11] [4].

The study "Monads for incremental computing"[6] performs the equality check in constant time, however the equality check is only on a shallow level. For example, when comparing a list for equality, the two lists are only equal if both lists are empty or both heads are equal, and the tail is the same modifiable. This can lead to recomputing the inputs while they are equal.

This paper solves these issues, by introducing the use of hash functions. The initial computation for computing the digests is the same as the previous equality, however every incremental computation is in constant time, because the digests are already computed. And it still compares the entire data structure of the input instead of a shallow comparison. Nevertheless, this still has drawbacks, because this solution has the risk of hash collisions, but with a large enough digest size the chance of having a hash collision is very small.

### 6.1.2 Storing the cached results

Besides, the comparison of equality of the inputs, the results of the given input also has to be stored. The idea of storing the results in a *Trie* is done by multiple studies [11] [18]. The implementation of the "Memo functions, polytypically!"[11] study, MemoTrie[7], uses the input as a key and the results as the value in the trie. This makes the performance of the `lookup` function dependent on

the size of the input, because the performance of the `lookup` on a trie is dependent on the size of the key. This becomes problematic when the input of a function is a large recursive data structure.

To solve this issue, the study "An efficient algorithm for type-safe structural diffing" by Miraldo and Swierstra uses the combination of digests and tries. As a result, the `lookup` function becomes constant time, because the size of the digest is fixed. This study is also the inspiration for using digests and tries.

### 6.1.3 Updating the input

Also, a difference between this paper and other studies are the updates of the input. This paper uses a Zipper to efficiently update the value in the data structure of the input and then updates the affected nodes. Alternatively, the study "Monads for incremental computing"[6] keeps track of all the modifications inside a Monad and then propagates the modifications when the `propogate` function is called. This is more efficient manner, because if two updates modify the same thing, the first update does not need to be calculated. Additionally, the data structure does not have to be navigated through, because of the modifiable references. However, for this paper the Zipper was a more easy to use technique, because it made it very easy to update the digests of the parents compared to using a Monad with modifiable references.

## 6.2 Future Work

The Thesis paper still has some topics that need further exploring. A few small topics would be:

- Implement a storage medium which does not rehash the key (because the key is already a hash).
- Implement a default cache replacement policy which generally works well for all algorithms.
- Implement the generation of the pattern synonyms using `TemplateHaskell`.
- Run the benchmarks in a more stable environment than a laptop.
- Benchmark the results against the `MemoTrie` package.

### 6.2.1 Support for Mutually Recursive Datatypes

The current implementation of the incremental algorithm only supports regular datatypes. This makes the possible datatypes the incremental algorithm can be used with, quite limited. To increase the amount of datatypes, we need to support mutually recursive datatypes. One big advantage of supporting mutually recursive datatypes is that then most abstract syntax tree (AST) of popular programming languages can be used with the incremental algorithm. So, for example, we can incrementally calculate the cyclomatic complexity metric over the AST of a programming language.

### 6.2.2 Implement the incremental algorithm using Sums-of-Products

The generic implementation of the incremental algorithm uses pattern functors. Pattern functors are a simple way to define generic functionality. However, the pattern functors have no restrictions on how they are combined. The Sums-of-Products represents the Haskell datatype better than the pattern functors, by only limiting the creation of sums of products. This can make it easier to implement the generic version of the incremental algorithm or add additional optimizations.

### 6.2.3 Benchmarking with real-world data

The current results presented in this paper are all synthetic benchmarks. This makes it easier to compare the results and make conclusions. However, it does not represent how well the algorithms actually perform in the real-world. A real-world example would be to compute a metric over public available code[1].

### 6.2.4 Support for a new input without changes

The incremental algorithm presented in this paper expects that the changes are given to them by an external system/process (e.g., a structure editor). This limits the way this algorithm can be used by other developers. To support more use-cases (e.g., a compiler), the incremental algorithm needs to support that when given a new input, without changes, it can still efficiently compute the result. A way to support this, is to perform a *diff* algorithm between the previous input and the current input and compute the changes between the two. Then using these changes update the previous input and compute the result using the incremental algorithm. An implementation of such *diff* algorithm can be found in the paper "On the Incremental Evaluation of Higher-Order Attribute Grammars" by Bransen in Section 4.6.3. However, this is certainly not faster than the current incremental algorithm and could be quite slower than using the non-incremental algorithm.

### 6.2.5 Prioritization

The simplest topic to research which adds a lot of value to this project is the generation of pattern synonyms using `TemplateHaskell`. Currently, the user has to define their own pattern synonyms which is quite developer unfriendly. The learning curve of `TemplateHaskell` is quite steep, but the eventual implementation is not complex. An example of using `TemplateHaskell` to generate pattern synonyms can be found at *Generics-MRSOP-TH*[17].

A more complex topic which adds a lot of value to the project is supporting mutually recursive datatypes. By supporting mutually recursive datatypes, the incremental algorithm can then be used for AST's of most programming languages. As a result, a new generic programming library needs to be used which supports mutually recursive datatypes (e.g., `multirec`[14]). This can take a lot of time,

[1] This does mean that the algorithm first needs to support mutually recursive datatypes.

## 6.3 Conclusion

We have created an algorithm for incremental computation over regular datatypes in Haskell. We have shown that the incremental algorithm performs faster than the non-incremental version when the data structure contains more than $10^3$ nodes. Also, the additional memory usage needed to store the cache for the incremental computation is negligible when correctly tuned. The better performance is accomplished by storing the digests for equality inside the recursive data structure and using a Zipper to efficiently update the digests when the data structure changes.

We introduced the pattern synonyms to improve the developer experience to almost the same level as the non-incremental implementation. The pattern synonyms could also be generated using `TemplateHaskell` to alleviate the amount of additional work for the developers.

We define possible cache addition and cache replacement policies to improve the performance/memory usage of the incremental algorithm. There are multiple policies defined to inspire the developers to find the fitting one for their use-case.

However, some difficulties still remain. First, the initial pass of the incremental algorithm is a lot slower than the non-incremental version. Therefore, the incremental algorithm needs to be performed a lot (with small changes), before being overall faster than the non-incremental version. Secondly, finding the correct policies for the best performance can be quite cumbersome. Third, the pattern synonyms have to be handwritten instead of being generated using `TemplateHaskell`.

# A

# Generic Programming

## A.1 Functor instances for Pattern Functors

```
instance Functor I where
  fmap f (I r) = I (f r)

instance Functor (K a) where
  fmap _ (K a) = K a

instance Functor U where
  fmap _ U = U

instance (Functor f, Functor g) => Functor (f :+: g) where
  fmap f (L x) = L (fmap f x)
  fmap f (R y) = R (fmap f y)

instance (Functor f, Functor g) => Functor (f :*: g) where
  fmap f (x :*: y) = fmap f x :*: fmap f y
```

# B

# Cache Management

## B.1 Implementation Recursion Depth

```
class HasDepth f where
  depth :: f (AFix g MemoInfo) -> Int

instance HasDepth (K a) where
  depth _ = 1

instance HasDepth U where
  depth _ = 1

instance HasDepth I where
  depth (I x) = let ph = getHasDepth (getAnnotation x) in 1 + ph

instance (HasDepth f, HasDepth g) => HasDepth (f :+: g) where
  depth (L x) = depth x
  depth (R x) = depth x

instance (HasDepth f, HasDepth g) => HasDepth (f :*: g) where
  depth (x :*: y) = max (depth x) (depth y)

instance (HasDepth f) => HasDepth (C c f) where
  depth (C x) = depth x
```

# C Results

## C.1 Minimum of 10 recursion depth

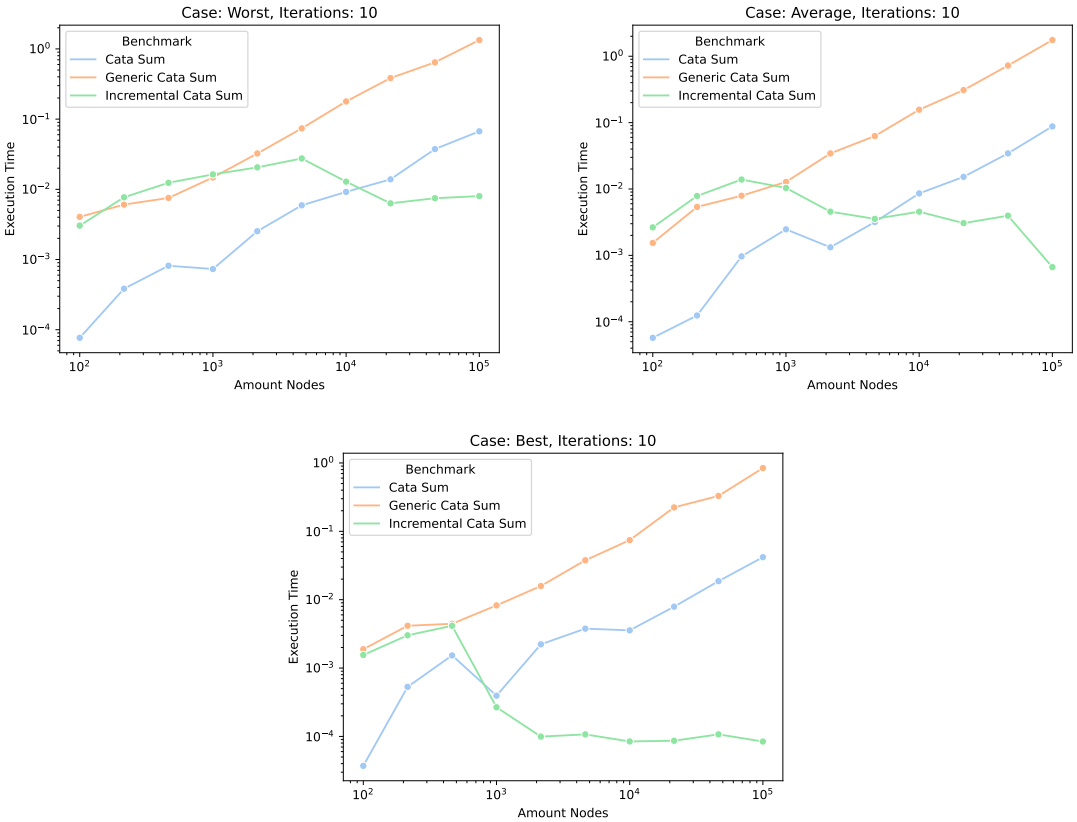


**Figure C.1:** The execution time over 10 executions for the Worst, Average and Best case, where the minimum recursion depth is 10.

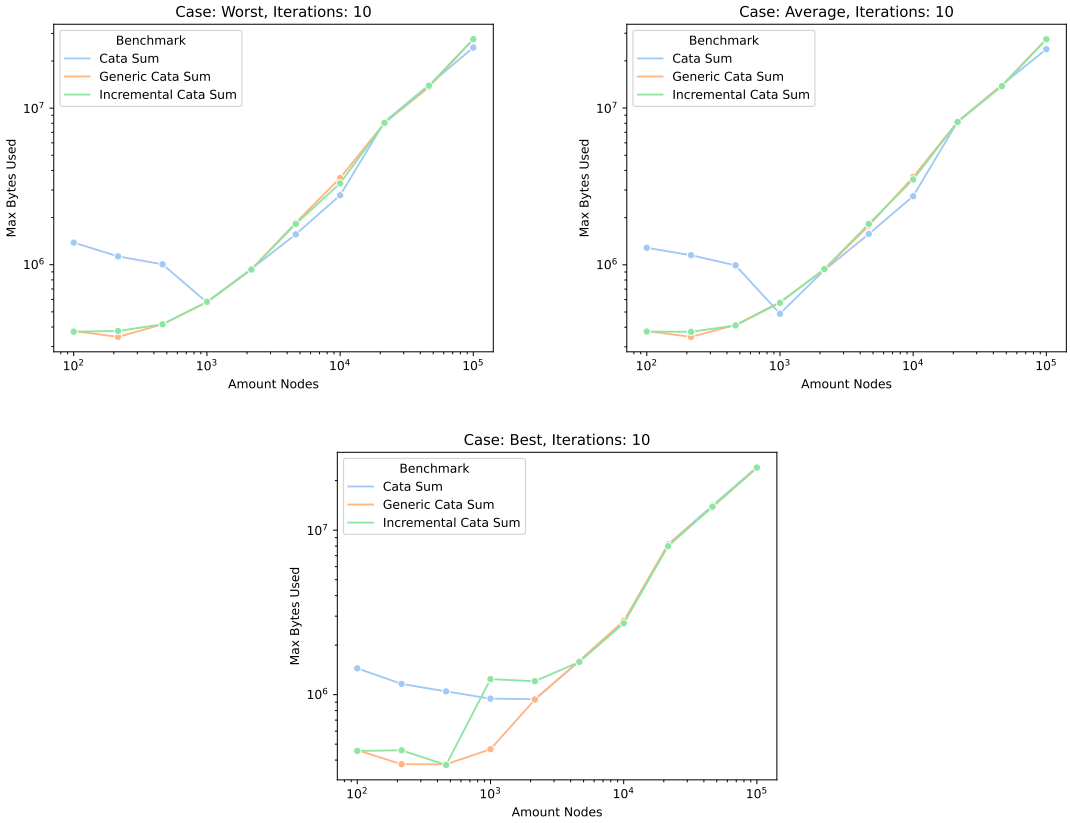


**Figure C.2:** The max-bytes-used over 10 executions for the Worst, Average and Best case, where the minimum recursion depth is 10.

## C.2 Individual benchmark results - Worst Case

### C.2.1 Linear trend line

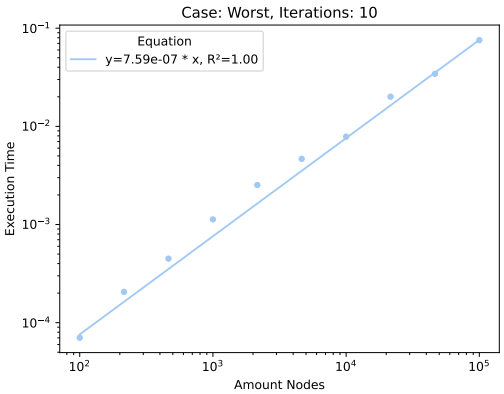


**Figure C.3:** Execution Time - Cata Sum

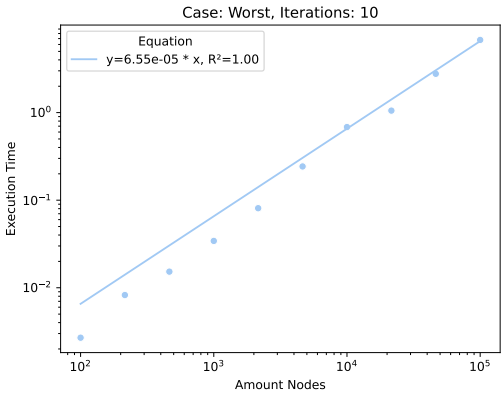


**Figure C.4:** Execution Time - Generic Cata Sum

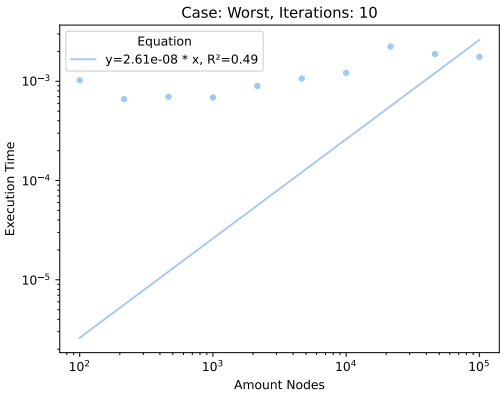


**Figure C.5:** Execution Time - Incremental Cata Sum

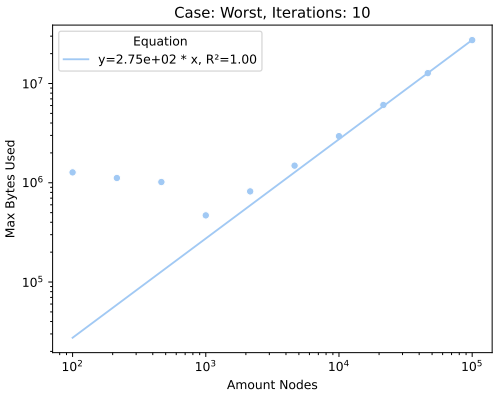


**Figure C.6:** Memory Usage - Cata Sum

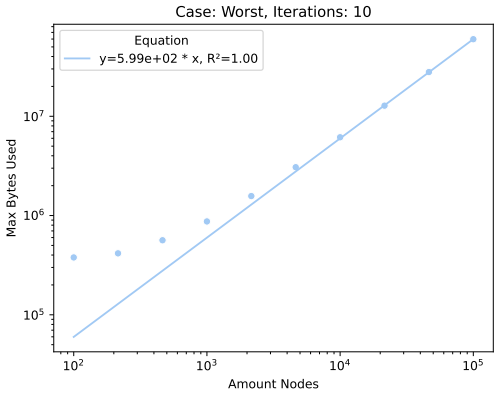


**Figure C.7:** Memory Usage - Generic Cata Sum

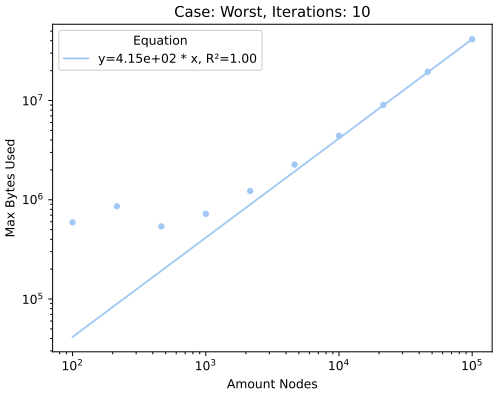


**Figure C.8:** Memory Usage - Incremental Cata Sum

### C.2.2 Logarithmic trend line

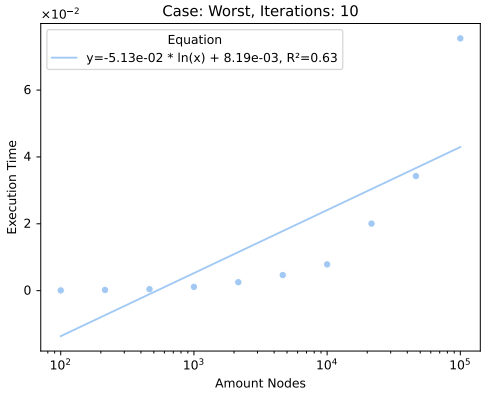


**Figure C.9:** Execution Time - Cata Sum

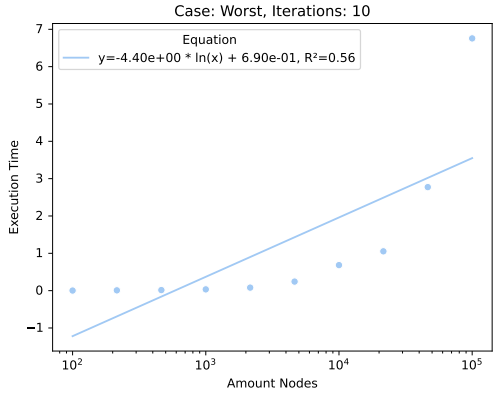


**Figure C.10:** Execution Time - Generic Cata Sum

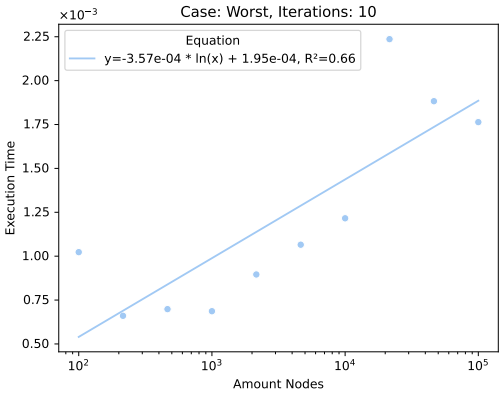


**Figure C.11:** Execution Time - Incremental Cata Sum

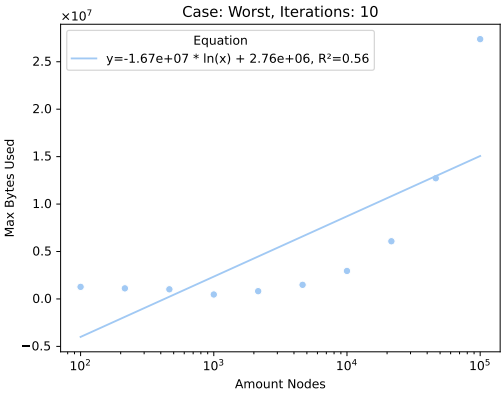


**Figure C.12:** Memory Usage - Cata Sum

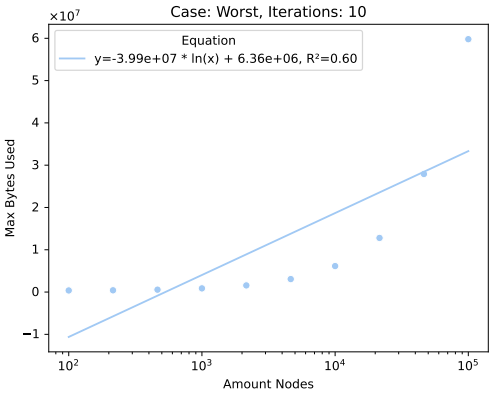


**Figure C.13:** Memory Usage - Generic Cata Sum

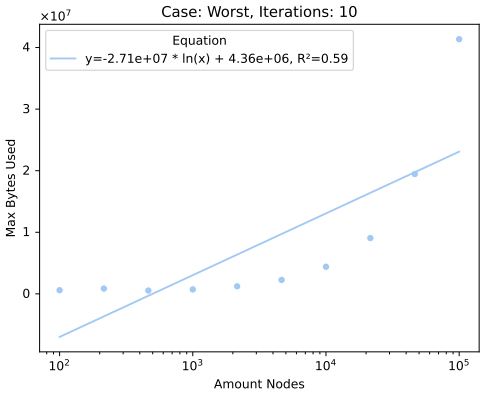


**Figure C.14:** Memory Usage - Incremental Cata Sum

## C.3 Individual benchmark results - Average Case

### C.3.1 Linear trend line

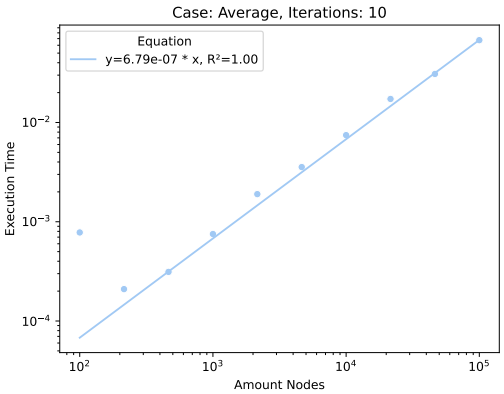


**Figure C.15:** Execution Time - Cata Sum

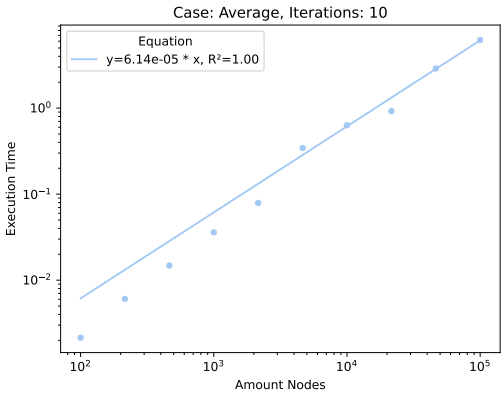


**Figure C.16:** Execution Time - Generic Cata Sum

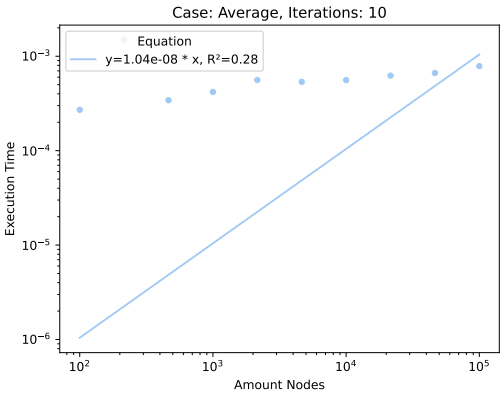


**Figure C.17:** Execution Time - Incremental Cata Sum

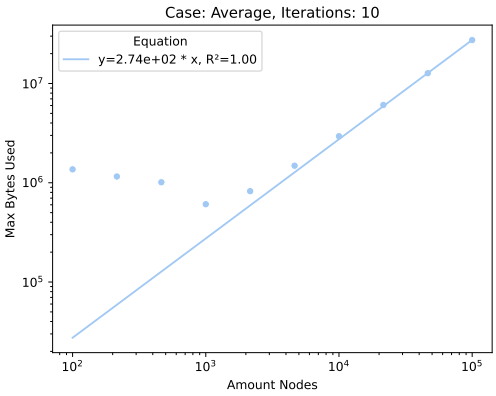


**Figure C.18:** Memory Usage - Cata Sum

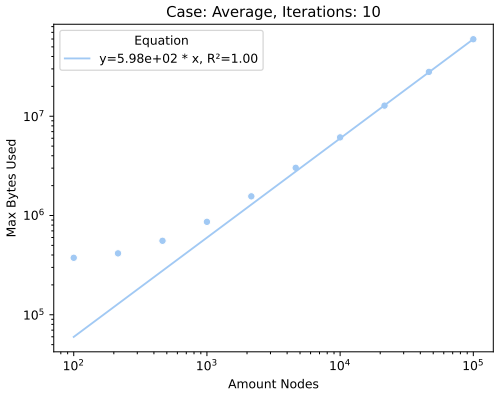


**Figure C.19:** Memory Usage - Generic Cata Sum

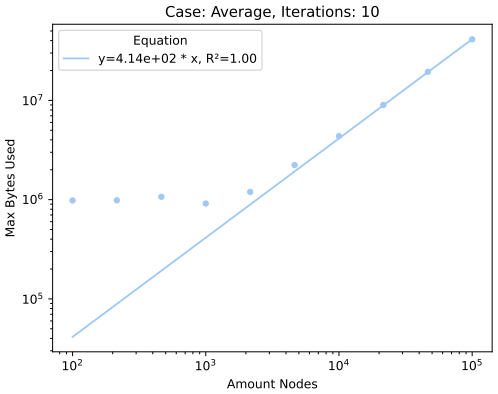


**Figure C.20:** Memory Usage - Incremental Cata Sum

### C.3.2 Logarithmic trend line

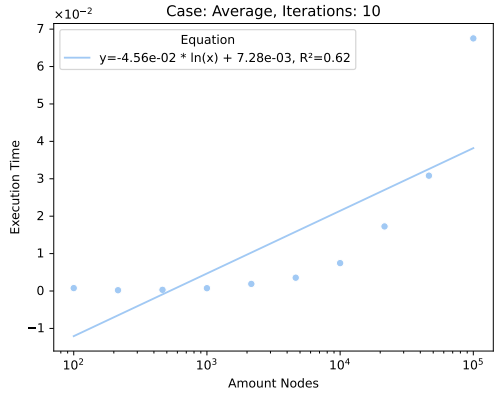


**Figure C.21:** Execution Time - Cata Sum

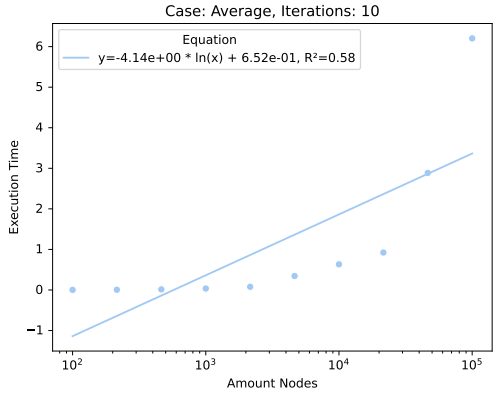


**Figure C.22:** Execution Time - Generic Cata Sum

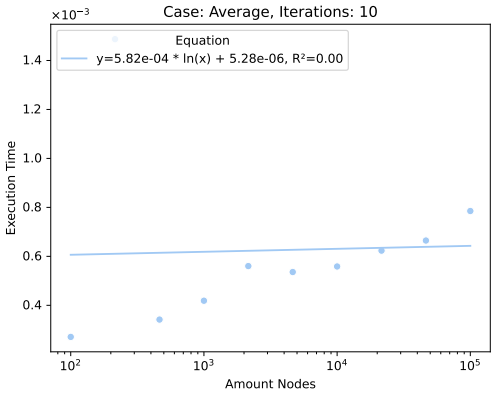


**Figure C.23:** Execution Time - Incremental Cata Sum

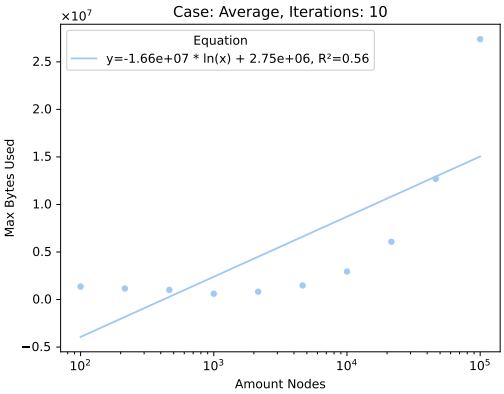


**Figure C.24:** Memory Usage - Cata Sum

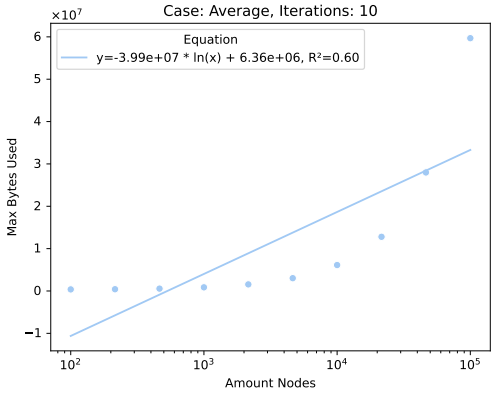


**Figure C.25:** Memory Usage - Generic Cata Sum

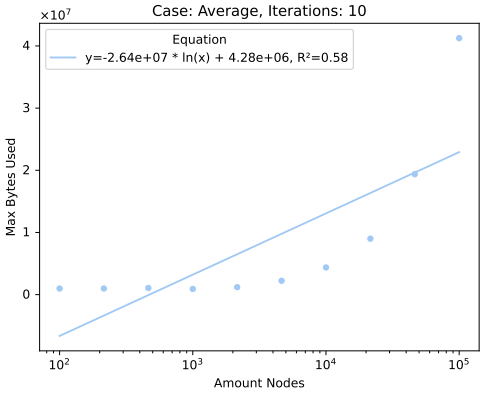


**Figure C.26:** Memory Usage - Incremental Cata Sum

## C.4 Individual benchmark results - Best Case

### C.4.1 Linear trend line

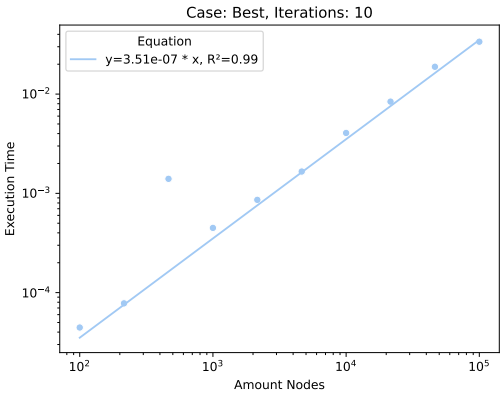


**Figure C.27:** Execution Time - Cata Sum

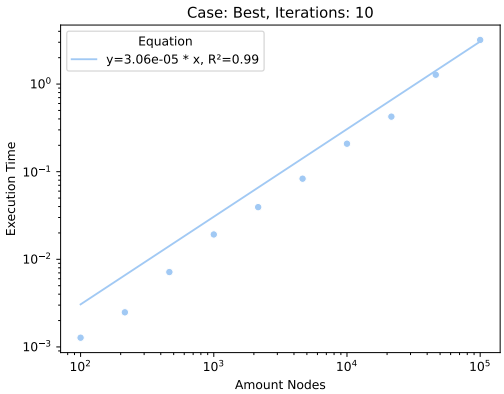


**Figure C.28:** Execution Time - Generic Cata Sum

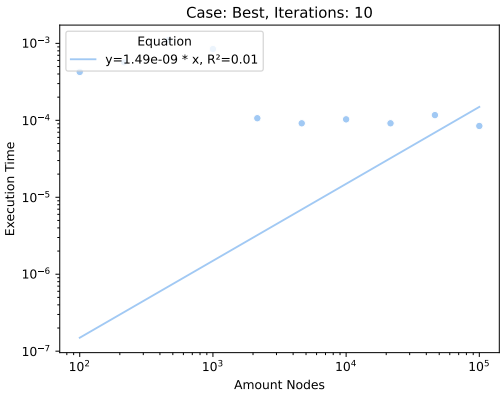


**Figure C.29:** Execution Time - Incremental Cata Sum

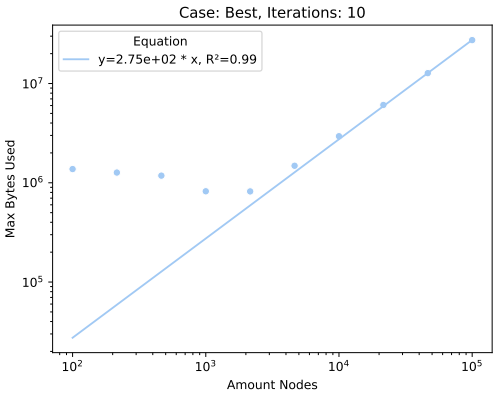


**Figure C.30:** Memory Usage - Cata Sum

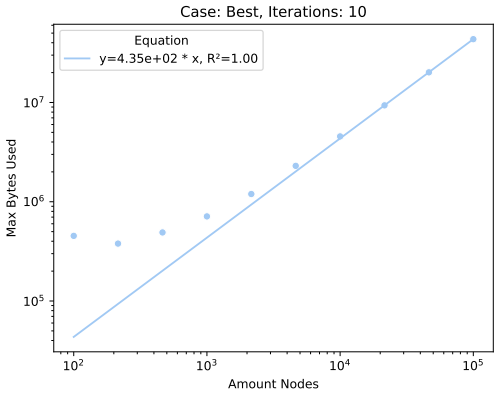


**Figure C.31:** Memory Usage - Generic Cata Sum

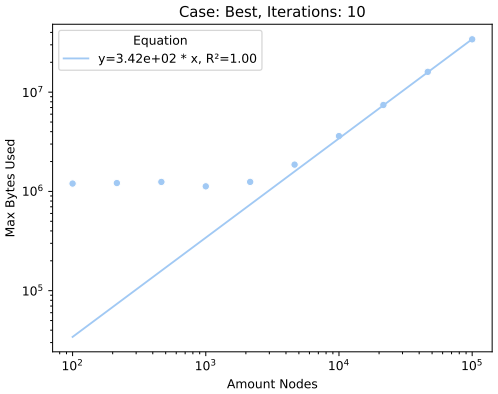


**Figure C.32:** Memory Usage - Incremental Cata Sum

### C.4.2 Logarithmic trend line

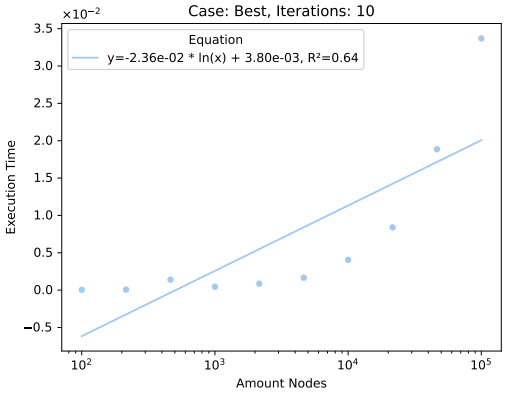


**Figure C.33:** Execution Time - Cata Sum

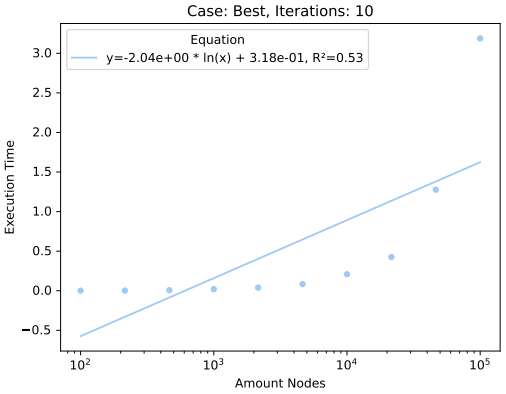


**Figure C.34:** Execution Time - Generic Cata Sum

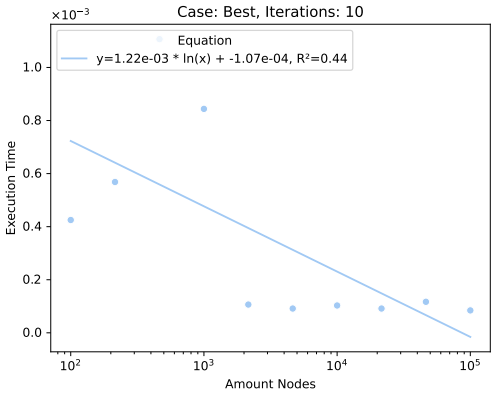


**Figure C.35:** Execution Time - Incremental Cata Sum

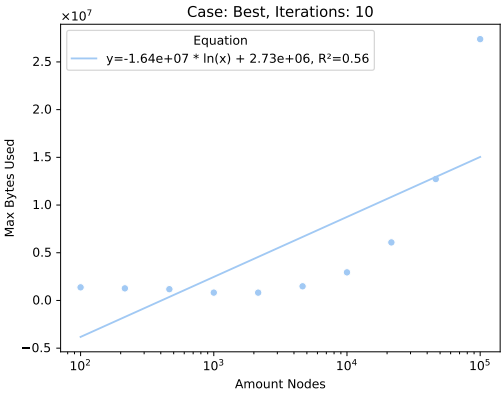


**Figure C.36:** Memory Usage - Cata Sum

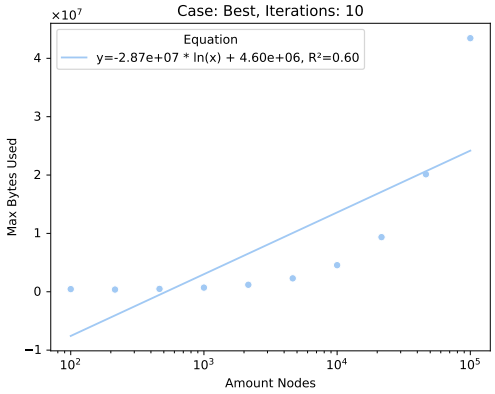


**Figure C.37:** Memory Usage - Generic Cata Sum

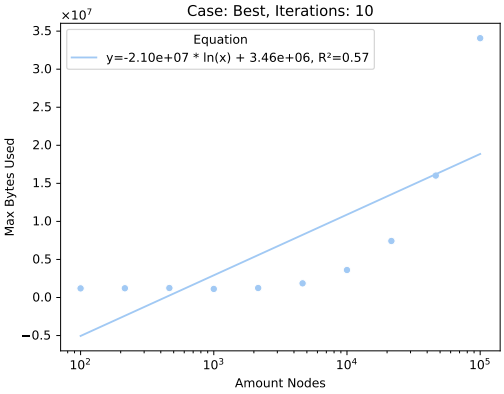


**Figure C.38:** Memory Usage - Incremental Cata Sum

# Bibliography


[1] Umut A Acar, Guy E Blelloch, and Robert Harper. "Adaptive functional programming". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 28.6 (2006), pp. 990–1034.

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

[3] Phil Bagwell. *Ideal hash trees*. Tech. rep. 2001.

[4] Jeroen Bransen. "On the Incremental Evaluation of Higher-Order Attribute Grammars". PhD thesis. Utrecht University, 2015.

[5] Jeroen Bransen and José Pedro Magalhaes. "Generic representations of tree transformations". In: *Proceedings of the 9th ACM SIGPLAN workshop on Generic programming*. 2013, pp. 73–84.

[6] Magnus Carlsson. "Monads for incremental computing". In: *Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*. 2002, pp. 26–35.

[7] Conal Elliott. *MemoTrie*. 2019. URL: `https://hackage.haskell.org/package/MemoTrie` (visited on July 28, 2022).

[8] Denis Firsov and Wolfgang Jeltsch. "Purely functional incremental computing". In: *Brazilian Symposium on Programming Languages*. Springer. 2016, pp. 62–77.

[9] Jeremy Gibbons. "Datatype-generic programming". In: *International Spring School on Datatype-Generic Programming*. Springer. 2006, pp. 1–71.

[10] HaskellWiki. *GHC/Type families*. 2021. URL: `https://wiki.haskell.org/GHC/Type_families` (visited on May 25, 2022).

[11] Ralf Hinze. "Memo functions, polytypically!" In: *Proceedings of the 2nd Workshop on Generic Programming, Ponte de*. Citeseer. 2000.

[12] Gérard Huet. "The zipper". In: *Journal of functional programming* 7.5 (1997), pp. 549–554.

[13] Oleg Kiselyov, Simon Peyton Jones, and Chung-chieh Shan. "Fun with type functions". In: *Reflections on the Work of CAR Hoare*. Springer, 2010, pp. 301–331.

[14] Andres Löh. *Generic programming for families of recursive datatypes*. URL: `https://hackage.haskell.org/package/multirec` (visited on May 10, 2022).

[15] Jose Pedro Magalhaes. *Generic programming library for regular datatypes*. URL: `https://hackage.haskell.org/package/regular` (visited on Apr. 28, 2022).

[16] Ralph C Merkle. "A digital signature based on a conventional encryption function". In: *Conference on the theory and application of cryptographic techniques*. Springer. 1987, pp. 369–378.

[17] Victor Miraldo and Alejandro Serrano. *Generic Programming with Mutually Recursive Sums of Products*. URL: `https://hackage.haskell.org/package/generics-mrsop` (visited on May 10, 2022).

[18] 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.

[19] Bryan O'Sullivan. *Robust, reliable performance measurement and analysis*. URL: `https://hackage.haskell.org/package/criterion` (visited on June 14, 2022).

[20] William Wesley Peterson and Daniel T Brown. "Cyclic codes for error detection". In: *Proceedings of the IRE* 49.1 (1961), pp. 228–235.

[21] Matthew Pickering et al. "Pattern synonyms". In: *Proceedings of the 9th International Symposium on Haskell*. 2016, pp. 80–91.

[22] Geoff Pike, Jyrki Alakuijala, and Software Engineering Team. *Introducing CityHash*. Apr. 11, 2011. URL: `https://opensource.googleblog.com/2011/04/introducing-cityhash.html` (visited on June 14, 2022).

[23] Jeff Preshing. *Hash Collision Probabilities*. 2011. URL: `https://preshing.com/20110504/hash-collision-probabilities` (visited on May 3, 2022).

[24] Austin Seipp. *Bindings to CityHash*. URL: `https://hackage.haskell.org/package/cityhash` (visited on June 15, 2022).

[25] Samir Talwar. *Transparent memoisation in Haskell with MemoTries*. URL: `https://monospacedmonologues.com/2022/01/memotries/` (visited on Aug. 2, 2022).

[26] GHC Team. *Running a compiled program*. URL: `https://downloads.haskell.org/~ghc/6.12.1/docs/html/users_guide/runtime-control.html` (visited on June 14, 2022).

[27] Johan Tibell. *Efficient hashing-based container types*. URL: `https://hackage.haskell.org/package/unordered-containers-0.2.19.1/docs/Data-HashMap-Strict.html` (visited on June 15, 2022).

[28] Edsko de Vries and Andres Löh. "True sums of products". In: *Proceedings of the 10th ACM SIGPLAN workshop on Generic programming*. 2014, pp. 83–94.

[29] Alexey Rodriguez Yakushev et al. "Generic programming with fixed points for mutually recursive datatypes". In: *ACM Sigplan Notices* 44.9 (2009), pp. 233–244.