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 2, 2022

Abstract

Incremental computation improves performance when the same computation is performed with slightly different input. An implementation of incremental computation is memoization. Memoization checks if the input of a function has already been computed and if it is return that *cached* result. However, comparing the input for equality to know if the result has already been computed is inefficient when folding over large recursive data structures. To improve the performance this paper introduces an algorithm which introduces digests of the internal structures and stores it inside the data structure (merkle tree), updating these digests using a Zipper when the data structure changes, writing a generic algorithm to support the class of regular datatypes. And all this while the developer experience is the same as writing the non-incremental algorithm in Haskell. In the end, we show that the performance is better than the non-incremental version with minimal extra memory usage, when correctly tuned with cache policies.

Contents

Todo list

1

Introduction

Incremental computation is an approach to improve performance by reusing results of a previously computed result when the both inputs are equal. There are multiple applications where incremental computation has a positive effect on: GUIs (e.g., DOM diffing), spreadsheets, attribute grammar evaluation, etc. An example, where incremental computation has significant effect on performance, would be 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 term which is associated with a function that caches the result of the function for the given input. A well-used Haskell package which implements memoization is the `MemoTrie` package[3].

```
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[18] of the Fibonacci algorithm using the `MemoTrie` package.

This works well when the comparison between inputs can be performed in a small amount of time. However, what if the input of that function is a large recursive data structure, for example a tree.

```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.

Then, to determine if the memoized inputs and the given input are equal, the entire tree needs to be traversed through. To improve the comparison of two 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.

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*[9]. 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

merkelize :: BinTree -> BinTreeH
merkelize (Leaf x)     = LeafH (hash ["Leaf", show x]) x
merkelize (Node l x r) = NodeH (hash ["Node", hl, hr, show x]) l' x r'
  where
    l' = merkelize l
    hl = getDigest l'
    r' = merkelize r
    hr = getDigest 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 needs to be rehashed, while only a small part of changes. The digests of the merkle tree only has to change if the node changes or one of the children nodes changes. In other words, only the nodes that changes and all its parents needs to be updated. To efficiently perform this in Haskell we use a technique named *Zipper*[6].

The Zipper keeps track, during the navigation from the root node to a specific node, of the path that has been traversed through the data structure. When the specific node is reached, we update the node with a given function and then update the digests accordingly to the new data structure. Therefore, we only have to update a part of the digest in the data structure.

```
data Direction = Lft | Rght
data Cxt = Cxt Direction BinTreeH Digest Int
type Loc = (BinTreeH, [Cxt])

left :: Loc -> Loc
left (NodeH d l x r, cs) = (l, (Cxt Lft r d x):cs)

right :: Loc -> Loc
right (NodeH d l x r, cs) = (r, (Cxt Rght l d x):cs)

up :: Loc -> Loc
up (l, (Cxt Lft r d x):cs)  = (NodeH d l x r, cs)
up (r, (Cxt Rght l d x):cs) = (NodeH d l x r, cs)
```

Figure 1.5: The implementation of a Zipper

Unfortunately, when we implement this functionality 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*.

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, 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, finally functionality for computing the result for a given function and cache with intermediate results.

However, this does mean that the given function which computes a result needs to use the representation types. The representation types 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*[14]. 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).

Finally, to keep the cache from growing too large for the available amount of memory, we show 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.

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 algorithm uses hashes for comparing if 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 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

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

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

Computing a value of a data structure can easily be defined in Haskell, but every time there is a small change in the `Tree`, the entire `Tree` needs to be recomputed. This is inefficient, because most of the computations have already been performed in the previous computation.

To prevent recomputation of already computed values, the technique memoization is introduced. Memoization is a technique where the results of computational intensive tasks are stored and when the same input occurs, the result is reused.

The comparison of two values in Haskell is done with the `Eq` typeclass, which implements the equality operator `(==) :: a -> a -> Bool`. So, 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.

Check if hash and hash values are correctly used

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 disadvantage of using hashes is *hash collisions*. Hash collisions happen when two different

pieces of data have the same 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*[16]. So, given a common hash function CRC-32[13], 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	5.06×10^9
128	2.17×10^{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*[17]. CityHash is a non-cryptographic hash function which works well for hash tables[15]. 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]
```

The hashes can then be used to efficiently compare two `Tree` data structures, without having to traverse the entire `Tree` data structure.

Add the implementation HashMap[20]

Looking up a value in a trie takes time proportional to the size of the value.

Explain HashMap and that it uses Tries, which leads to constant lookups

To keep track of the intermediate results of the computation, we store the results in a `HashMap`[20]. A `HashMap` is an implementation of mapping a *hashable* key to a value. In our next example the `Digest` is the key and the value is the intermediate result.

```
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[10] inspired the use of the Merkle Tree. A Merkle Tree is a data structure which integrates the hashes within the data structure.

2.1 Merkle Tree (`TreeH`)

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 l@(Leaf x) = LeafH (hash l) x
merkle (Node l x r) = NodeH h l' x r'
  where
    h = hash ["Node", x, getHash l', getHash r']
    l' = merkle l
    r' = merkle r
```

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

```haskell
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 hashes which are impacted by the changes. Which means that only the hashes 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 data structure which can be used to efficiently update the data structure, namely the Zipper[6].

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[6] 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`).

```haskell
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 `TreeH`

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 nodes 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 hash of that parent and recursively update the parents hashes until we are at the top of the data structure. Otherwise, return the modified locations, because all the other hashes 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
```

```haskell
update f dirs t = leave $ updateLoc f l'
  where
    l' = applyDirs dirs (enter t)
```

3

Datatype-Generic Programming

Compare paper "A Lightweight Approach to Datatype-Generic Rewriting"

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 custom classes. Therefore, we use *Datatype-Generic Programming*[4] to define functionality for a large class of datatypes.

In this chapter, we introduce Datatype-Generic Programming, also known as *generic programming* or *generics* 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`[8]. Here the generic representation of a datatype is called a *pattern functor*. A pattern functor is a stripped-down version of a data type, 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 functors defined in `regular` are 5 core pattern functors and 2 meta information pattern functors. The core pattern functors describe the datatypes. The meta information pattern functors only contain information (e.g., constructor name) but not any structural information.

Use "representation types" instead of "pattern functor" for these datatypes

```
data U r          = U                -- Empty constructors
data I r          = I r              -- Recursive call
```

```
data K a r       = K a                -- Constants
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 regular 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 determine the length for every pattern functor `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 pattern functors return zero. The `U` pattern functor returns zero, because it does not contain any children. The `K` pattern functor 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 pattern functors are quite similar. The sums pattern functor recurses into the specified choice. The product pattern functor 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 pattern functors, a function can be defined using the generic length function. By first, converting the datatype into a generic representation, then calling `glength` given recursively itself, 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.

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

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

```
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.

```
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

Add a description of what a universe is

A different way of describing datatypes in a generic representation, besides pattern functors, are *Sums of Products*[21] (SOP). SOP is a generic representation with additional constraints 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. The SOP universe is described using *codes* 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 of kind `[[*]]` can then be interpreted to describe Haskell datatypes of kind `*`. To define a code, the tick mark `` ` `` is used to lift the list to a type-level.

```
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, the additional constraints on the generic representation makes the SOP universe size comparatively bigger than pattern functors. Therefore, it is more complex to extend the SOP than for pattern functors.

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 of many programming languages, we need to support datatypes which can recurse over different datatypes, namely mutually recursive datatypes.

```haskell
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 *indexed fixed-points*[22]. The indexed fixed-points works by creating a type family φ 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 a lot more complexity.

4

Generic Implementation

4.1 Regular

The `regular` generic programming library was chosen, because it has the smallest universe size compared to other libraries. Therefore, implementing the generic implementation is less complex. However, `regular` does not support *Sums of Products* and *mutually recursive datatypes*, but we expect that the results will be meaningful without supporting these features.

Write about why Regular is chosen

The first step of the 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 hash (`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 generic datatypes. We introduce a typeclass named `Hashable` which defines a function `hash`, which converts the `f` datatype into a `Digest` (also known as a *hash value*).

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

digest :: Show a => a -> Digest
digest = digestStr . show -- converts a string into a hash value
```

The Hashable instance of `U` is simple. The `digest` function is used to convert the constructor name `U` into a `Digest`. The `K` also uses the `digest` function to convert the constructor name into a `Digest`, but it also calls `digest` 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 _ = digest "U"
```

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

The instances for `:+:`, `:*:` and `C` 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 [digest "L", hash x]
  hash (R x) = digestConcat [digest "R", hash x]

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

instance (Hashable f) => Hashable (C c f) where
  hash (C x) = digestConcat [digest "C", hash x]
```

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 generic representation, into a generic representation 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 intermediate

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 we first need to give the function a `HashMap Digest a`. To simplify the use of `cataMerkleState`, we define a function `cataMerkle`, 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, we have all the necessary functionality defined to write a function over the generic representation and automatically generate all the intermediate 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[2].

The context (`Ctx`) is implemented using a type family[1][7]. 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.

```
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

[1] "*Type families are to vanilla data types what type class methods are to regular functions.*"[5]

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 intermediate value can be stored. Unfortunately, we are limited by the amount of memory there is available on the machine. Therefore, the amount of intermediate values needs to be limited. There are two suggestions of limiting the amount of intermediate values with *Cache Addition Policies* and *Cache Replacement Policies*. Cache addition policies are policies which indicate if an intermediate value can be added to the cache. And the cache replacement policies are policies which remove intermediate values 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 we then know how many times we have to go into recursion before reaching a leaf. 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 intermediate values and could potentially lead to a speed-up, because the lookup in the HashMap could take longer than the calculation itself.

The recursion depth can be determined in the same pass as when the hashes 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 }
```

```haskell
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.

```haskell
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
```

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 intermediate values from the cache. 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 intermediate values. If the intermediate value is true with the current cache replacement policy, it gets discarded. Otherwise, the intermediate value stays in the cache.

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

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

-- Add the CacheInfo to the intermediate values
cataMerkle :: (f a -> a) -> AFix f MemoInfo
           -> State (HashMap Digest (a, CacheInfo)) a
```

[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."[1]

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

Possible cache replacement policies:

- 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.

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*[14].

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. However, writing all the patterns for all pattern functors is an arduous task. Luckily, we can use `TemplateHaskell` to generate the pattern synonyms[3].

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

[3] An example of using `TemplateHaskell` to generate pattern synonyms can be found at *Generics-MRSOP-TH*

```
  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 intermediate 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`[12]. 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 gather for 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[19] 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. (1) The worst case updates the lowest left leaf to a new leaf. (2) The average case updates a node in the middle with a new leaf. (3) 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

For all the benchmarks, 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 performance 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 discrepancy in the range between $10^2 - 10^3$ is probably because the amount of nodes is too low to get stable results.

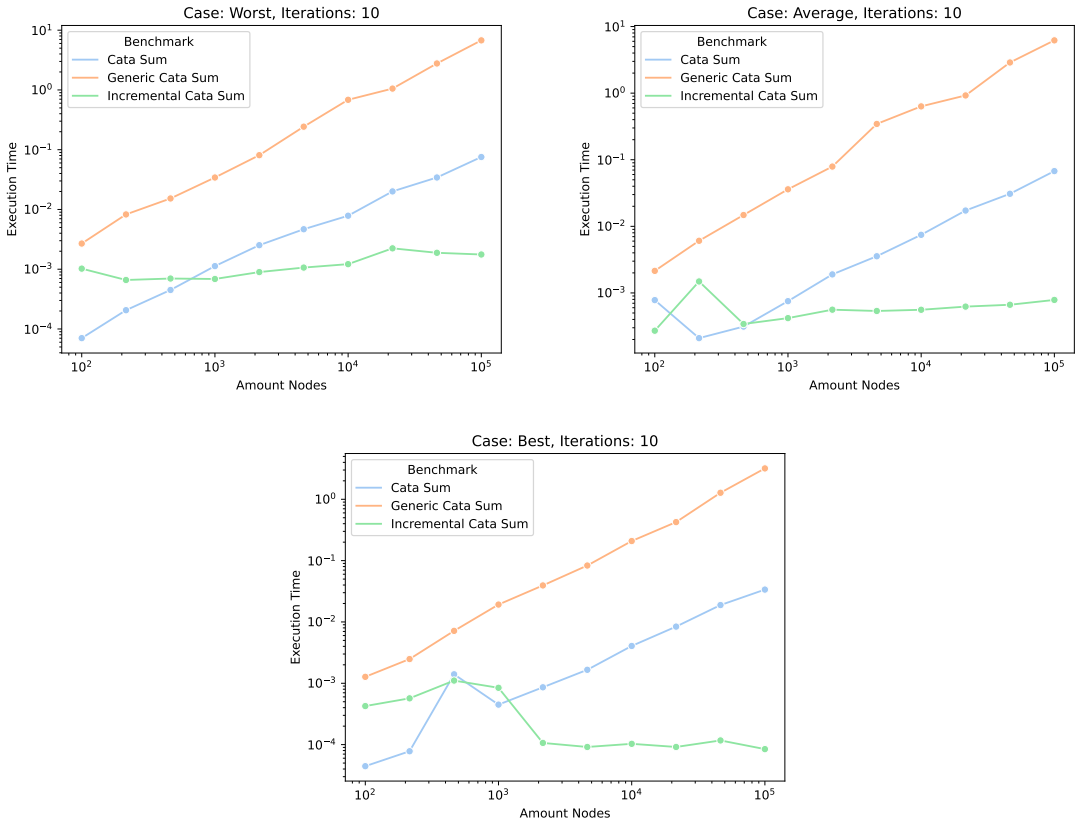

Figure 5.1: The execution time over 10 executions for the Worst, Average and Best case.

5.2.2 Memory Usage

The memory usage results are all 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.

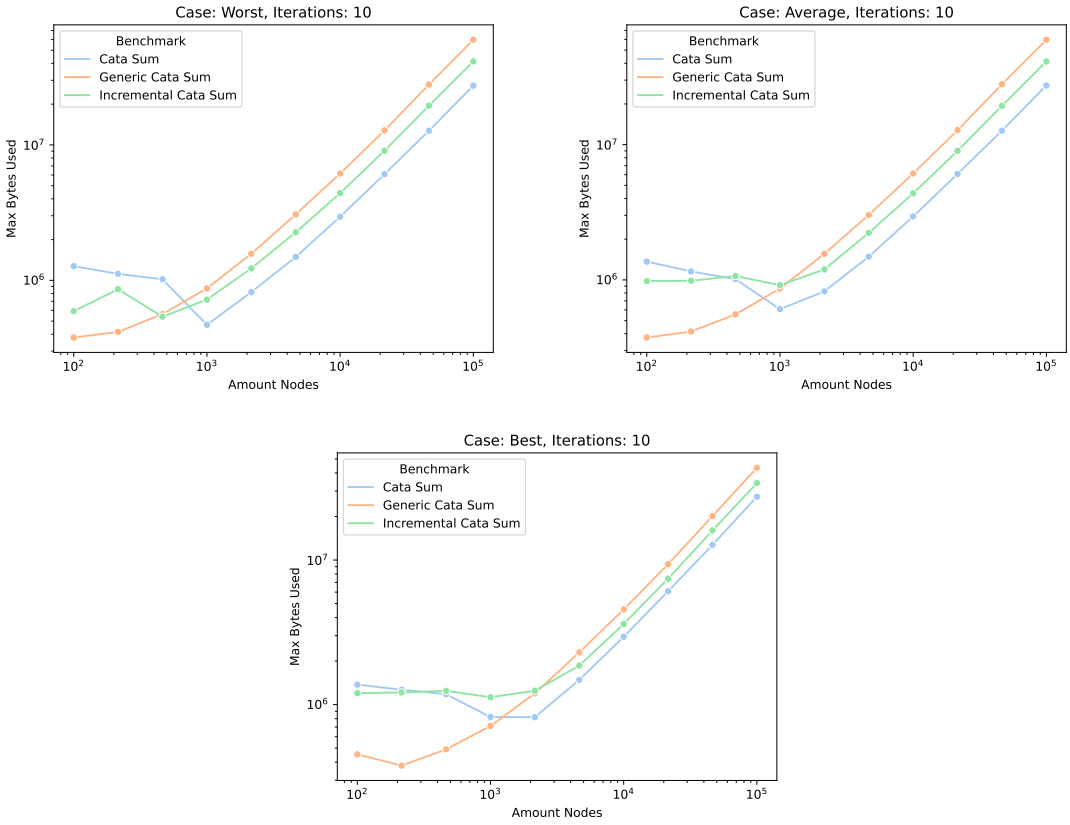

Figure 5.2: The max-bytes-used over 10 executions for the Worst, Average and Best case.

5.2.3 Comparison Cache Addition Policies

As expected the memory usage is lower than when all the intermediate values 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

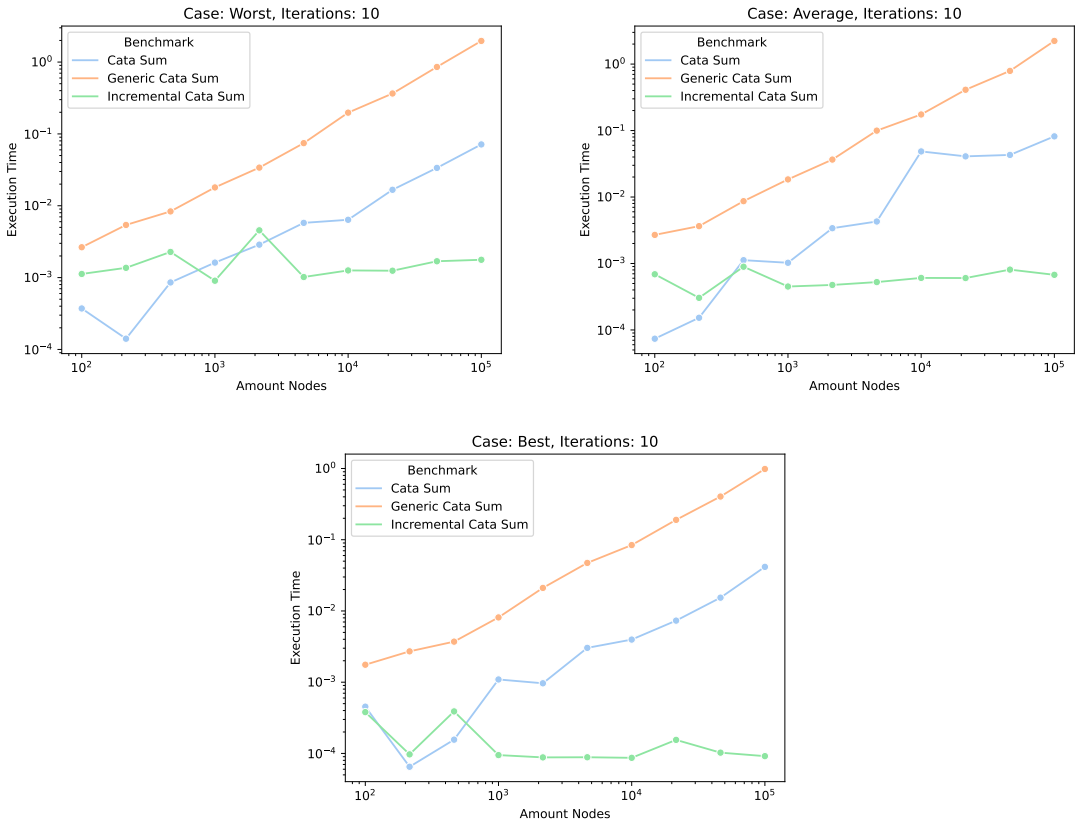

Figure 5.3: The execution time over 10 executions for the Worst, Average and Best case, where the minimum recursion depth is 5.

Memory Usage

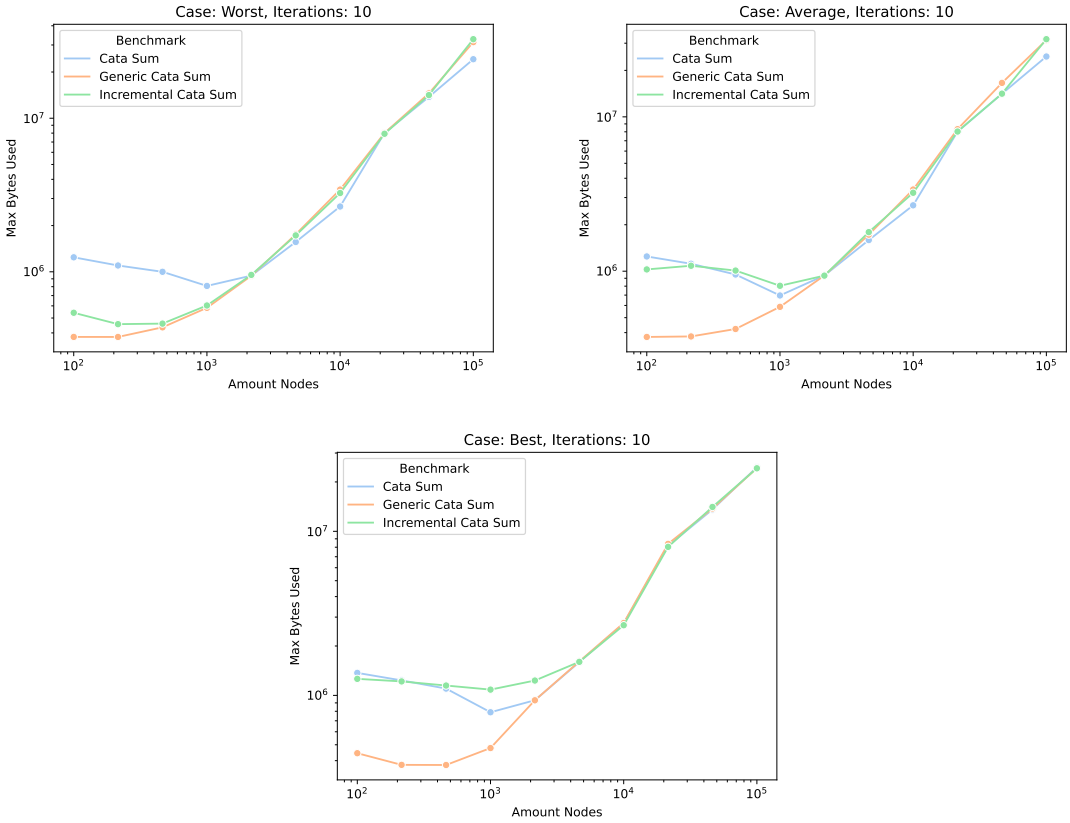

Figure 5.4: The max-bytes-used over 10 executions for the Worst, Average and Best case, where the minimum recursion depth is 5.

6

Conclusion

We have created an algorithm for incremental computation over recursive datatypes in Haskell. We have shown that the algorithm performs faster than the non-incremental version after 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 hashes for equality inside the recursive data structure and using a Zipper to efficiently update the hashes 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 can also be generated using `TemplateHaskell` to elevate 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 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 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.

6.1 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.

6.1.1 Support for Mutually Recursive Datatypes

The current implementation of the algorithm only supports Regular datatypes. This makes the possible datatypes this 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 AST of popular programming languages can be used with the algorithm. So, for example, we can incrementally calculate the cyclomatic complexity metric over the AST of a programming language.

6.1.2 Implement the algorithm using Sums-of-Products

The generic implementation of the 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 algorithm or add additional optimizations.

6.1.3 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].

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

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

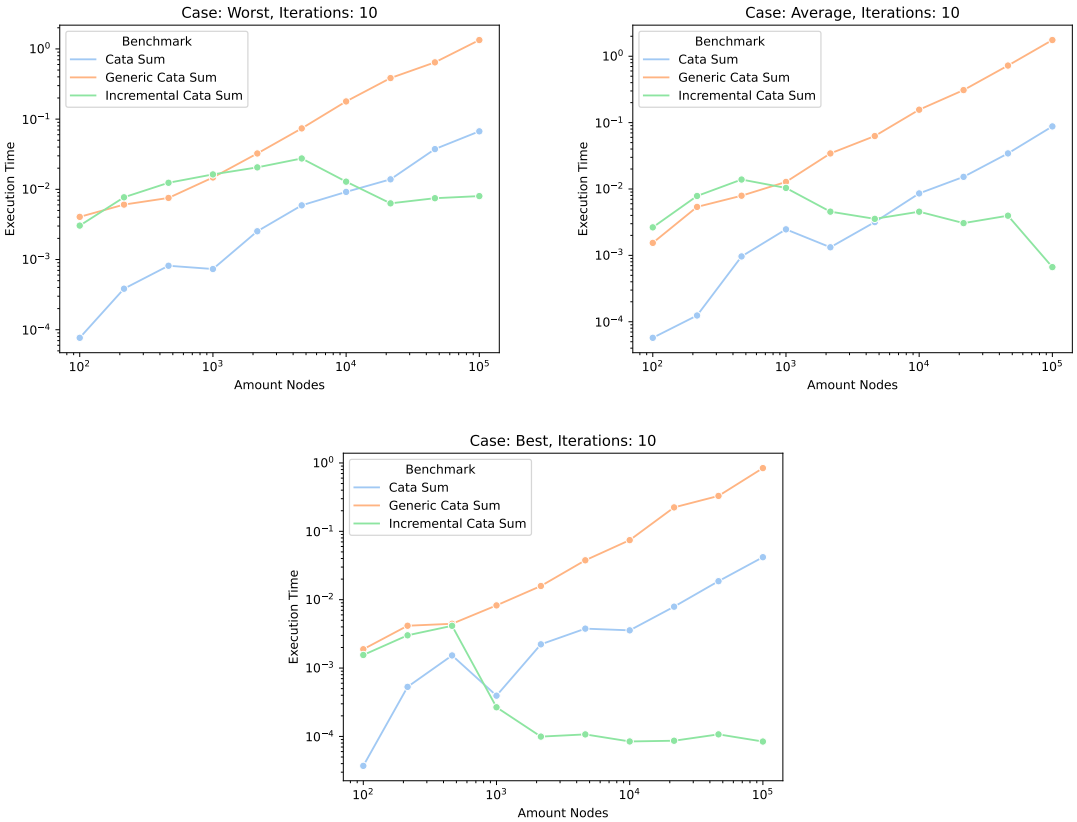

Figure C.1: The execution time over 10 executions for the Worst, Average and Best case, where the minimum recursion depth is 10.

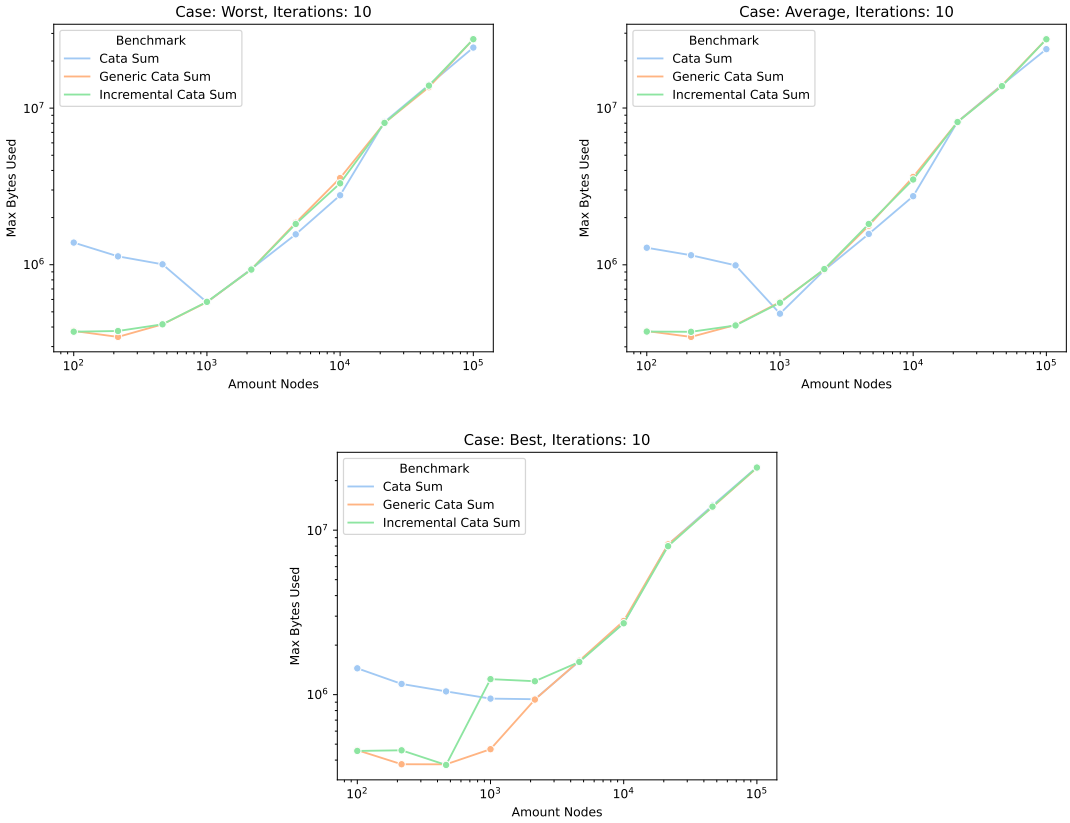

Figure C.2: The max-bytes-used over 10 executions for the Worst, Average and Best case, where the minimum recursion depth is 10.

C.2 Individual benchmark results - Worst Case

C.2.1 Linear trend line

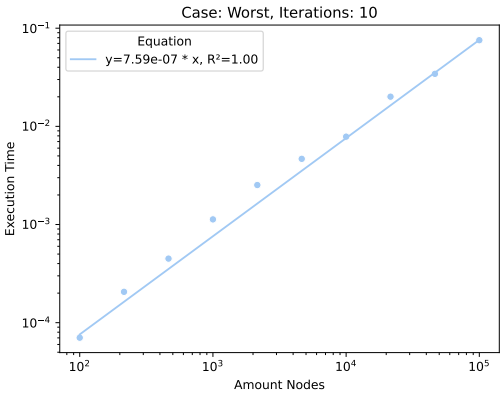

Figure C.3: Execution Time - Cata Sum

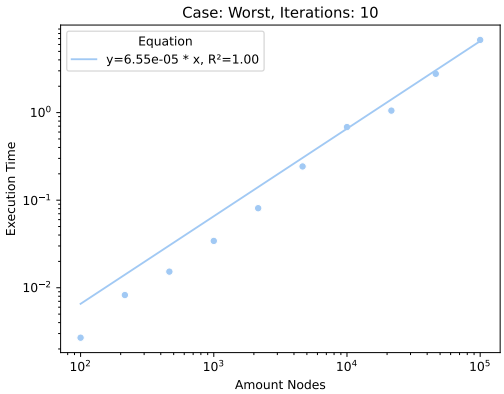

Figure C.4: Execution Time - Generic Cata Sum

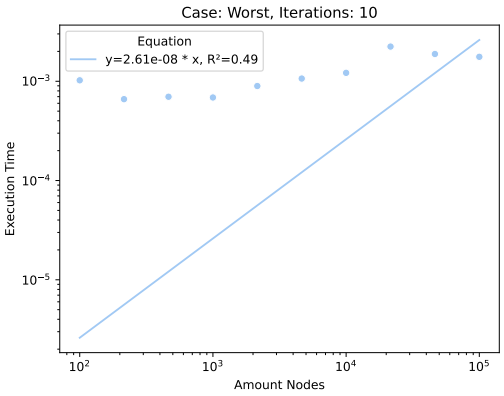

Figure C.5: Execution Time - Incremental Cata Sum

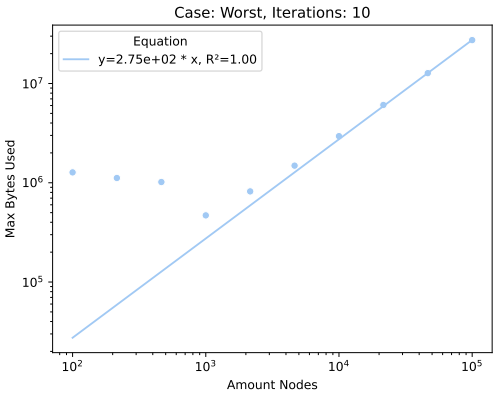

Figure C.6: Memory Usage - Cata Sum

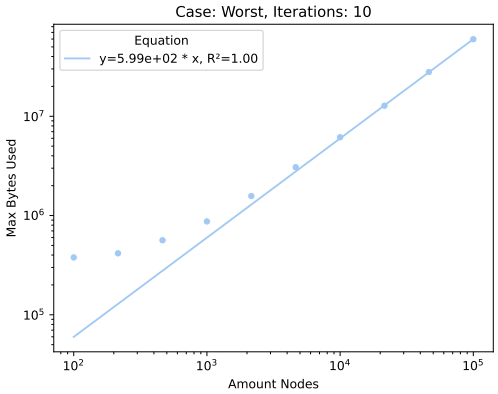

Figure C.7: Memory Usage - Generic Cata Sum

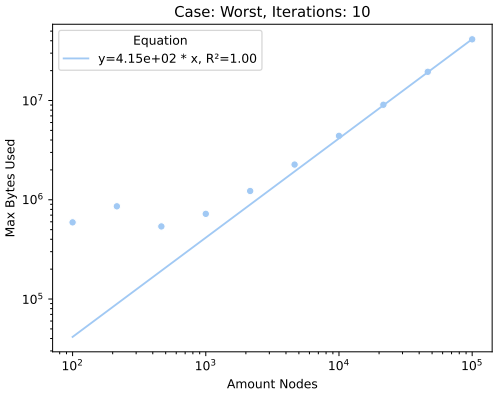

Figure C.8: Memory Usage - Incremental Cata Sum

C.2.2 Logarithmic trend line

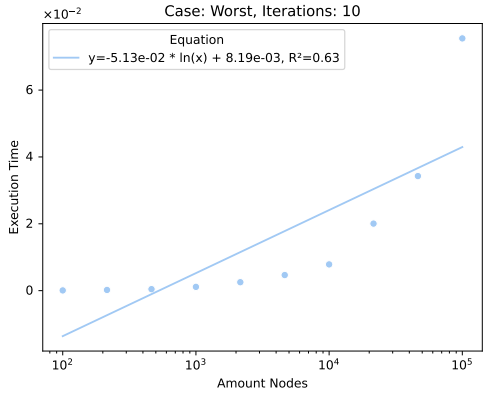

Figure C.9: Execution Time - Cata Sum

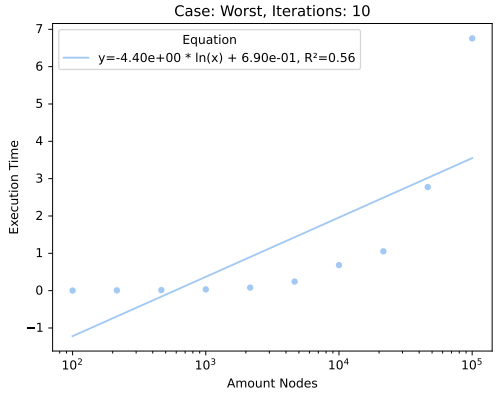

Figure C.10: Execution Time - Generic Cata Sum

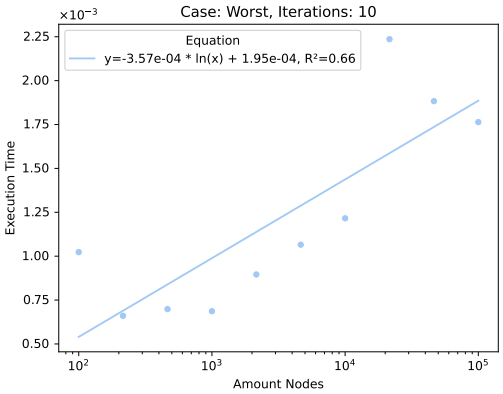

Figure C.11: Execution Time - Incremental Cata Sum

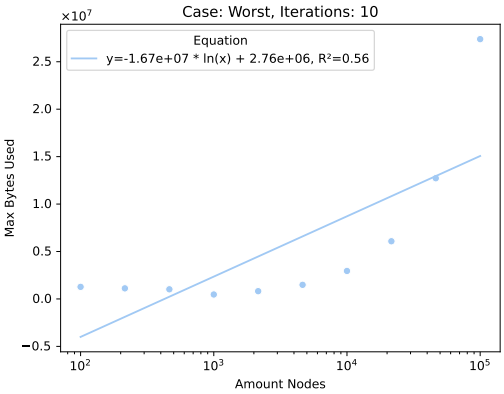

Figure C.12: Memory Usage - Cata Sum

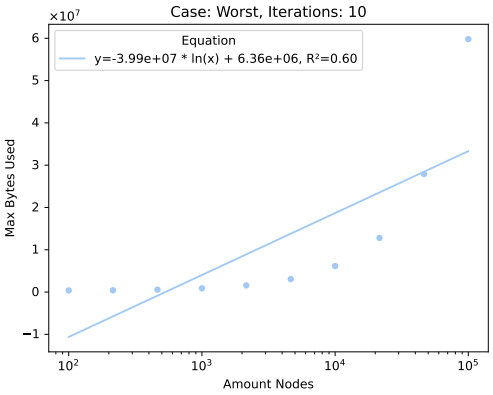

Figure C.13: Memory Usage - Generic Cata Sum

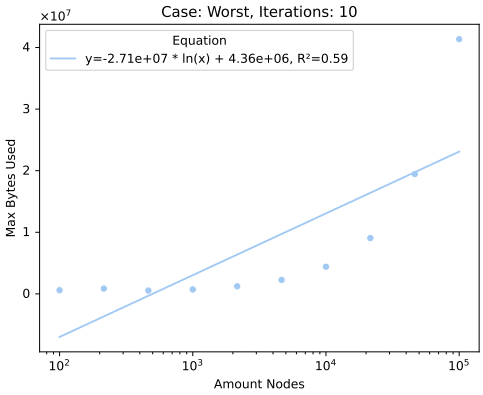

Figure C.14: Memory Usage - Incremental Cata Sum

C.3 Individual benchmark results - Average Case

C.3.1 Linear trend line

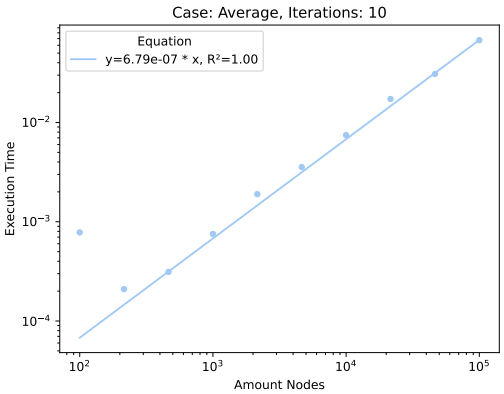

Figure C.15: Execution Time - Cata Sum

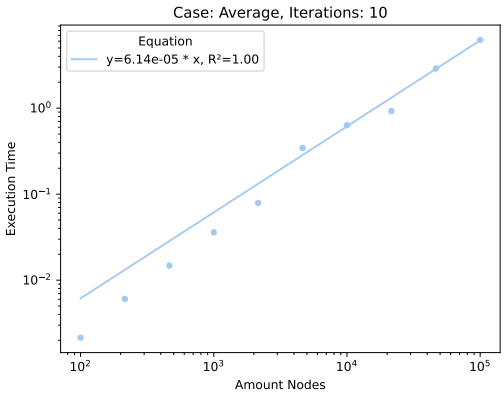

Figure C.16: Execution Time - Generic Cata Sum

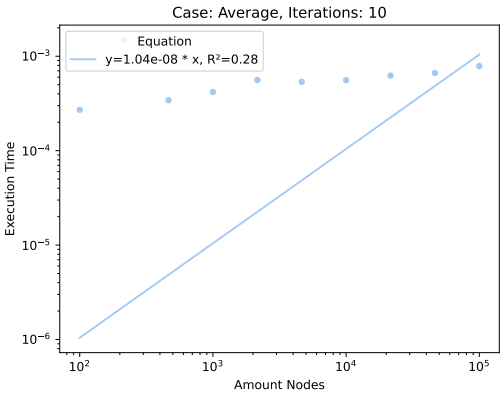

Figure C.17: Execution Time - Incremental Cata Sum

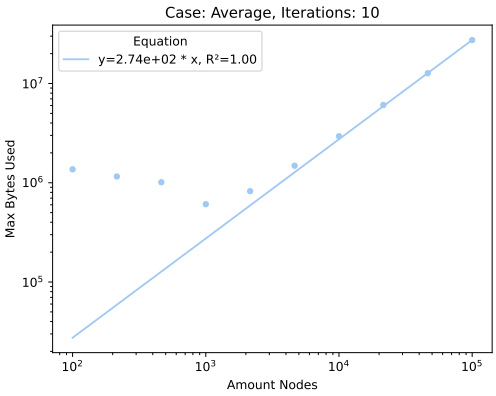

Figure C.18: Memory Usage - Cata Sum

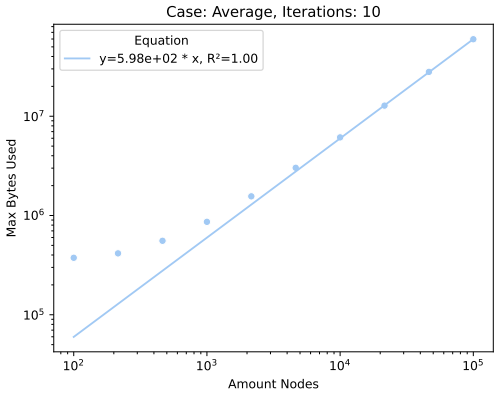

Figure C.19: Memory Usage - Generic Cata Sum

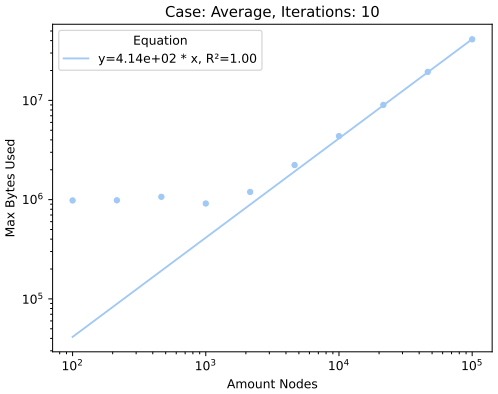

Figure C.20: Memory Usage - Incremental Cata Sum

C.3.2 Logarithmic trend line

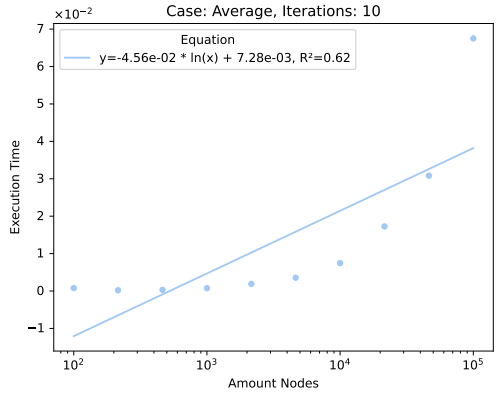

Figure C.21: Execution Time - Cata Sum

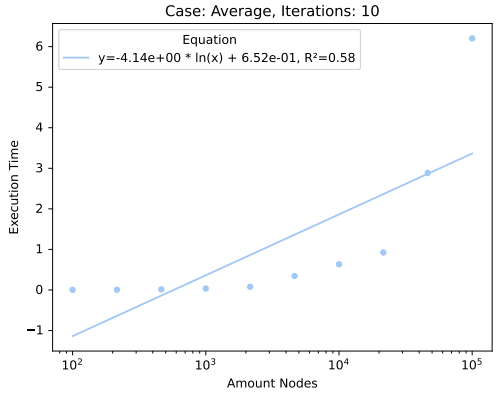

Figure C.22: Execution Time - Generic Cata Sum

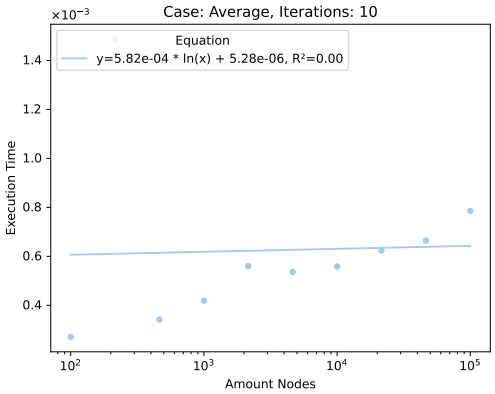

Figure C.23: Execution Time - Incremental Cata Sum

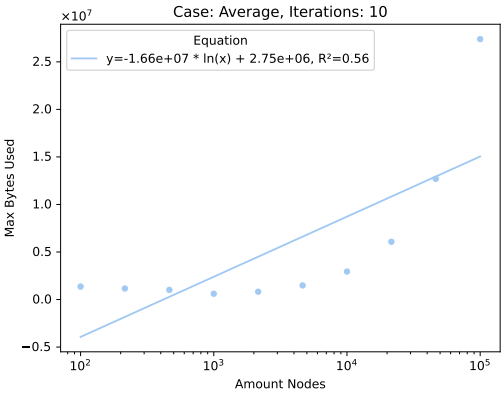

Figure C.24: Memory Usage - Cata Sum

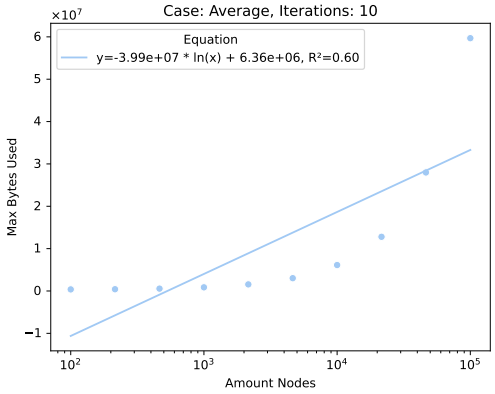

Figure C.25: Memory Usage - Generic Cata Sum

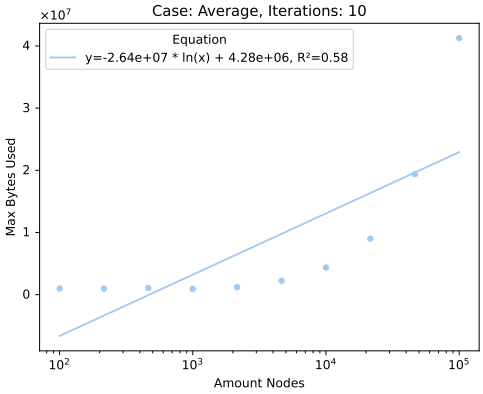

Figure C.26: Memory Usage - Incremental Cata Sum

C.4 Individual benchmark results - Best Case

C.4.1 Linear trend line

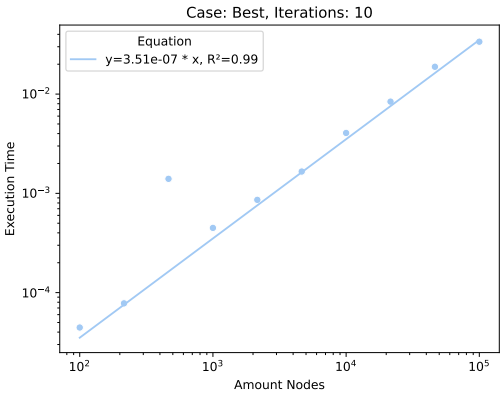

Figure C.27: Execution Time - Cata Sum

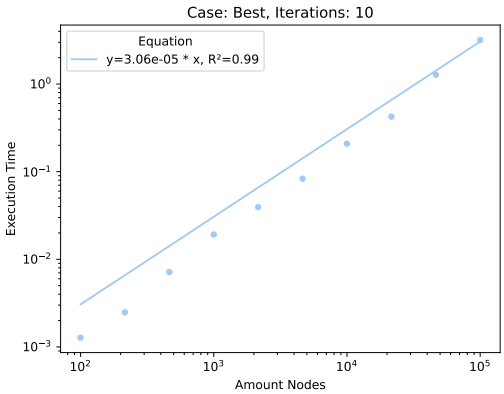

Figure C.28: Execution Time - Generic Cata Sum

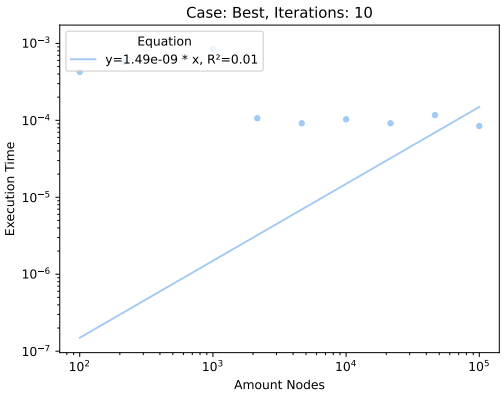

Figure C.29: Execution Time - Incremental Cata Sum

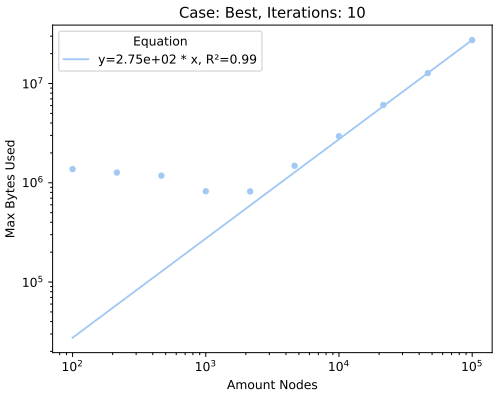

Figure C.30: Memory Usage - Cata Sum

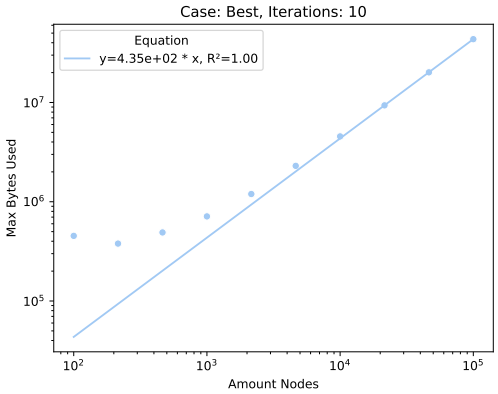

Figure C.31: Memory Usage - Generic Cata Sum

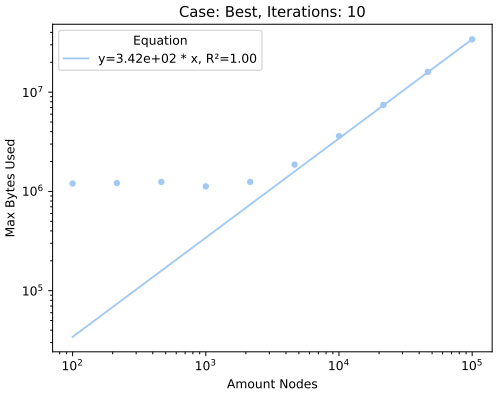

Figure C.32: Memory Usage - Incremental Cata Sum

C.4.2 Logarithmic trend line

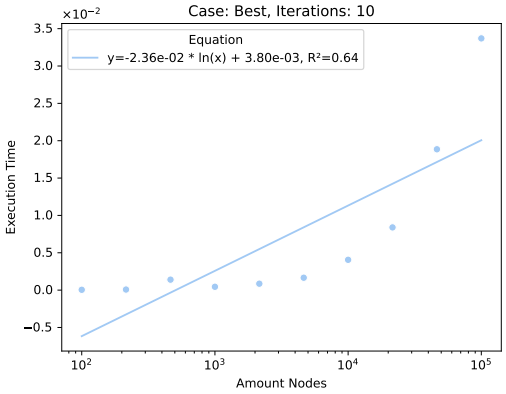

Figure C.33: Execution Time - Cata Sum

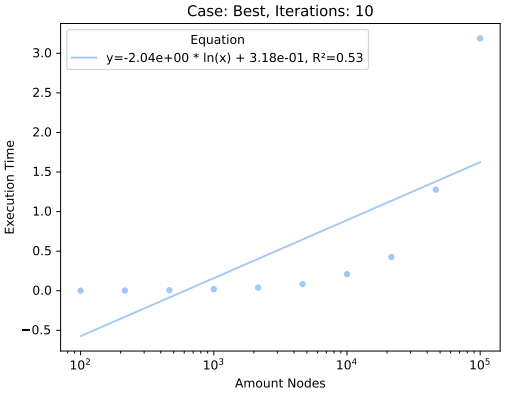

Figure C.34: Execution Time - Generic Cata Sum

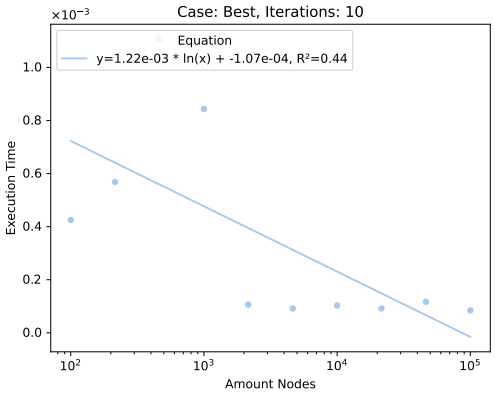

Figure C.35: Execution Time - Incremental Cata Sum

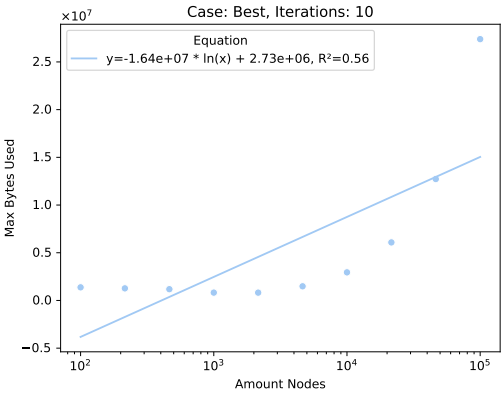

Figure C.36: Memory Usage - Cata Sum

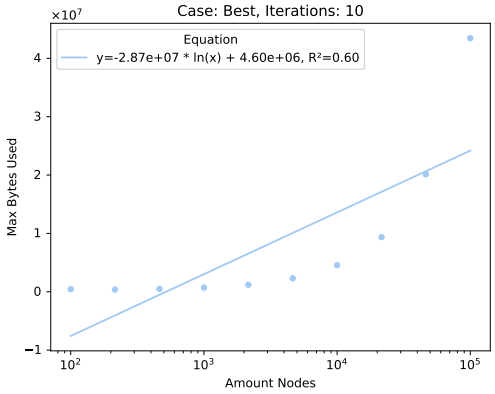

Figure C.37: Memory Usage - Generic Cata Sum

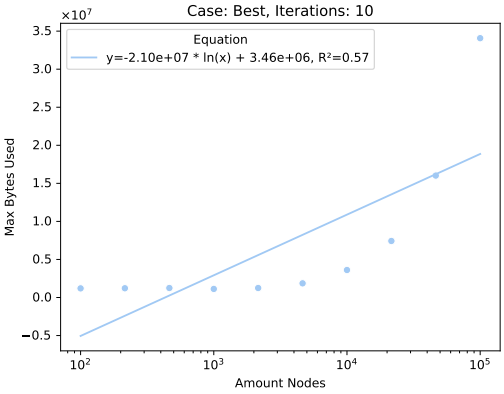

Figure C.38: Memory Usage - Incremental Cata Sum

Bibliography

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

[2] 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.

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

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

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

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

[7] 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.

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

[9] 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.

[10] 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.

[11] Thomas van Noort et al. "A Lightweight Approach to Datatype-Generic Rewriting". In: (2008).

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

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

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

[15] 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).

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

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

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

[19] 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).

[20] 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).

[21] 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.

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