



**Utrecht
University**

Computing Science MSc Thesis

Incremental Computation for Algebraic Datatypes in Haskell

Author

Jort van Gorkum (6142834)

Supervisors

Wouter Swierstra

Trevor McDonell

Faculty of Science

Department of Information and Computing Sciences

Programming Technology

April, 2022

Write abstract

1	Introduction	4
1.1	Motivation	4
1.2	Contributions	4
1.3	Research Questions	4
2	Specific Implementation	5
2.1	Merkle Tree (TreeH)	7
2.2	Zipper	8
2.2.1	Zipper TreeH	9
3	Prototype Implementation	10
3.1	Generic programming in Haskell	10
3.1.1	Pattern Functors vs Sums of products	10
3.1.2	Mutually recursive datatypes	10
3.1.3	Comparison Generic Libraries	10
3.2	Prototype language	10
3.3	HashMap vs Trie	12
4	Generic Implementation	13
4.0.1	Regular	13
4.1	Complexity	13
4.2	Memory Strategies	14
4.3	Pattern Synonyms	14
5	Experiments	15
5.1	Execution Time	15
5.2	Memory Usage	17
5.3	Comparison Memory Strategies	19
6	Conclusion and Future Work	20
6.1	Conclusion	20
A	Implementation Memo Cata	21
A.1	Definition Generic Datatypes	21
A.2	Implementation Hashable	21
A.3	Implementation Merkle	22
A.4	Implementation Cata Merkle	22
A.5	Implementation Zipper Merkle	23
B	Regular	24
B.1	Zipper	24

Todo list

Write abstract	1
Write a motivation	4
Write the contributions	4
Write the research questions	4
Write about the bad side of using hashes	5
Write about the reuse of the Map	6
Maybe add the more efficient implementation of merging maps?	6
Add a visual example	8
Introduce generic programming and how it works	10
Describe the differences between defining generic data types	10
Describe what mutually recursive datatypes are and why do we need to know about it . .	10
Describe the use of the Zipper and how the hashes are updated	12
Write a piece about the comparison of storing it in a HashMap or a Trie datastructure . .	12
Write about Hdiff and the use of Trie datastructure	12
Write about why Regular is chosen	13
Write about the implementation of Regular and what had to change compared to the prototype language	13
Describe for every function used the complexity and what leads to the complete complexity	13
Describe multiple memory strategies for keeping memory usage and execution time low .	14
Write about paper selective memoization	14
Explain Pattern Synonyms	14

1

Introduction

1.1 Motivation

Write a motivation

1.2 Contributions

Write the contributions

1.3 Research Questions

Write the research questions

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.

Write about the bad side of using hashes

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.

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 of the disadvantages 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. However, common hash functions have such a low chance of getting a hash collision it is negligible.

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

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. To keep track of the intermediate results of the computation, we store the results in a `Map`. A `Map`, also known as a dictionary, is an implementation of mapping a key to a value. In our next example the `Hash` is the key and the value is the intermediate result.

```
sumTreeInc :: Tree Int -> (Int, Map Hash 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
```

Write about the reuse of the `Map`

Maybe add the more efficient implementation of merging maps?

```
sumTreeIncMap :: Map Hash Int -> Tree Int -> (Int, Map Hash 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 subset of the data structure for every execution is time-consuming

and is unnecessary because most of the data structure stays the same. So, the Merkle Tree concept is used, which is inspired by the work of Miraldo and Swierstra[2]. The Merkle Tree integrates the hashes with the data structure.

2.1 Merkle Tree (TreeH)

First we introduce a new datatype `TreeH`, which contains a `Hash` 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 Hash a
             | NodeH Hash (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 `Map`, without computing the hashes every time the `sumTreeIncH` function is called.

```
sumTreeIncH :: TreeH Int -> (Int, Map Hash 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`[1] data structure.

2.2 Zipper

Add a visual example

The Zipper data structure works by keeping track of how the data structure is being traversed through. The Zipper keeps track by using a *context*. The context is the inverse of the direction which the data structure is traversed through. Meaning that when we traverse to the left side of the `Tree` the right side of the `Tree` gets stored in the context.

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

Then combining the `Tree` and the `Cxt`, we can traverse through the `Tree` and keeping the *context* of how we got to the current point.

```
type Loc a = (Tree a, Cxt a)
```

To use the `Loc`, first we need to create a `Cxt`.

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

Using the `Loc`, we can define multiple function on how to traverse through the `Tree`.

```
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)
```

When we get to the desired point in the `Tree` which needs to be updated, we call the `modify` function.

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

Then to get the entire updated tree back, we only need to call the `up` function indefinitely until the `Loc` is at the top.

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

2.2.1 Zipper TreeH

To implement the Zipper for `TreeH`, the same setup can be used as in the previous section, but when modifying the `TreeH` we also need to update all the parent nodes. First the `Cxt` needs to be updated to additionally store the hashes. Then the direction functions need to be updated to also store the Hash in the `Cxt`.

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

When modifying the value in the `TreeH`, the parents nodes hashes also needs to be updated.

```
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)

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

Prototype Implementation

3.1 Generic programming in Haskell

Introduce generic programming and how it works

3.1.1 Pattern Functors vs Sums of products

Describe the differences between defining generic data types

3.1.2 Mutually recursive datatypes

Describe what mutually recursive datatypes are and why do we need to know about it

3.1.3 Comparison Generic Libraries

3.2 Prototype language

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

The definition of the pattern functor only leads to shallow recursion. Meaning that pattern functor can only be used to observe a single layer of recursion. To apply a function over the complete data structure, deep recursion is used. To implement deep recursion, the fix point is introduced.

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

The fix point is then used to describe the recursion of the datatype on a type-level basis. Using pattern functors and fix point most of the Haskell datatypes can be represented. For example:

```

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

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

```

Because the generic representation of the Haskell datatypes can be represented using pattern functors, we can use Functors. Using the Functor class a `cata` function can be defined, which is a generic fold function.

```

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

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

```

To store the intermediate results of `cata`, we want the structure of the data to be hashed. This way we can easily compare if the data structure has changed over time, without completely recomputing the resulting digests. To do this, first a fix point is introduced which additionally stores the digest.

```

type Merkle f = Fix (f :+: K Digest)

```

Then to convert the fix point to a fix point containing the structural digest, the `Merkelize` class is introduced.

```

class Hashable f where
  hash :: Hashable g => f (Fix g) -> (f :+: K Digest) (Fix (g :+: K Digest))

merkleG :: Hashable f
  => f (Fix (g :+: K Digest)) -> (f :+: K Digest) (Fix (g :+: K Digest))
merkleG f = f :+: K (hash f)

merkle :: Hashable f => Fix f -> Merkle f
merkle = In . merkleG . fmap merkle . from

```

Using the new fix point with its structural digest, a new `cata` function can be defined which can store its intermediate values in a `Map Digest a`.

```

cataMerkleState :: (Functor f, Traversable f, Container c, Show (c a), Show a)
  => (f a -> a) -> Merkle f -> State (c a) a
cataMerkleState alg (In (Pair (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

cataMerkle :: (Traversable f, Container c, Show (c a), Show a)
           => (f a -> a) -> Merkle f -> (a, c a)
cataMerkle alg t = runState (cataMerkleState alg t) empty

```

Zipper

Describe the use of the Zipper and how the hashes are updated

3.3 HashMap vs Trie

Write a piece about the comparison of storing it in a HashMap or a Trie datastructure

Write about Hdiff and the use of Trie datastructure

4

Generic Implementation

4.0.1 Regular

Write about why Regular is chosen

Write about the implementation of Regular and what had to change compared to the prototype language

```
newtype K a r    = K { unK :: a }      -- Constant value
newtype I r      = I { unI :: r }      -- Recursive value
data U r         = U                  -- Empty Constructor
data (f :+: g) r = L (f r) | R (g r)  -- Alternatives
data (f **: g) r = f r **: g r         -- Combine
data C c f r     = C { unC :: f r }    -- Name of a constructor
data S l f r     = S { unS :: f r }    -- Name of a record selector

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

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

4.1 Complexity

Describe for every function used the complexity and what leads to the complete complexity

4.2 Memory Strategies

Describe multiple memory strategies for keeping memory usage and execution time low

Write about paper selective memoization

4.3 Pattern Synonyms

Explain Pattern Synonyms

```
{-# COMPLETE Leaf_, Node_ #-}

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))

cataSum :: MerklePF (Tree Int) -> (Int, M.Map Digest Int)
cataSum = cataMerkle
  (\case
    Leaf_ x      -> x
    Node_ l x r -> l + x + r
  )
```

5

Experiments

5.1 Execution Time

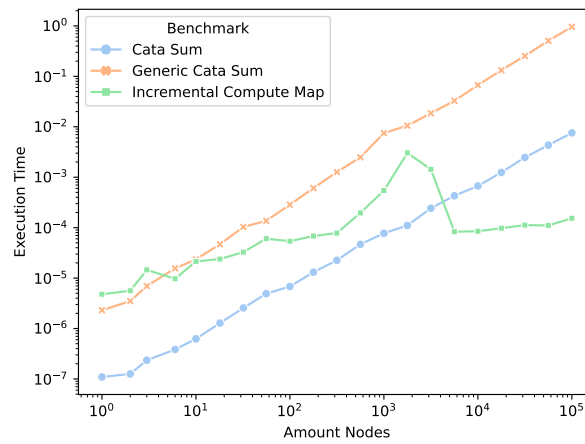


Figure 5.1: Overview execution time

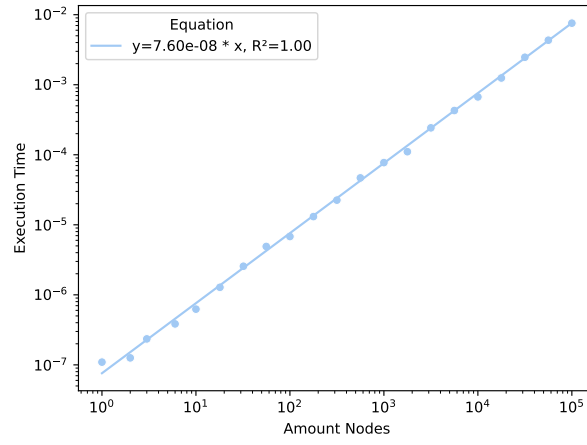


Figure 5.2: Execution time for Cata Sum

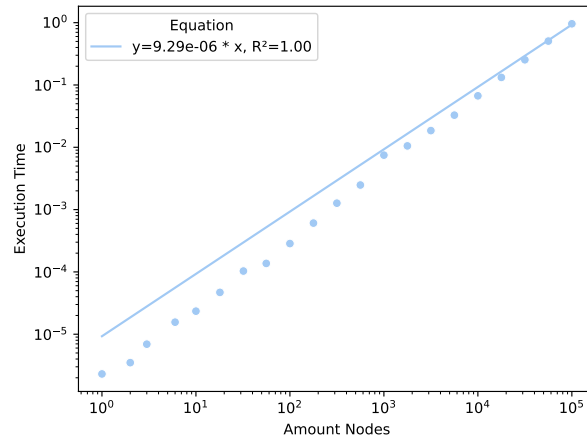


Figure 5.3: Execution time for Generic Cata Sum

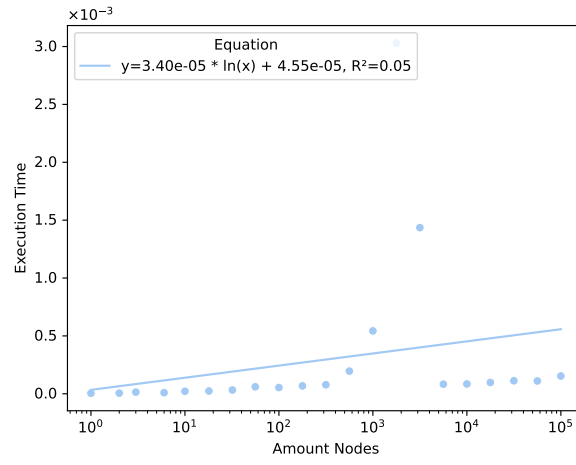


Figure 5.4: Execution time for Incremental Cata Sum

5.2 Memory Usage

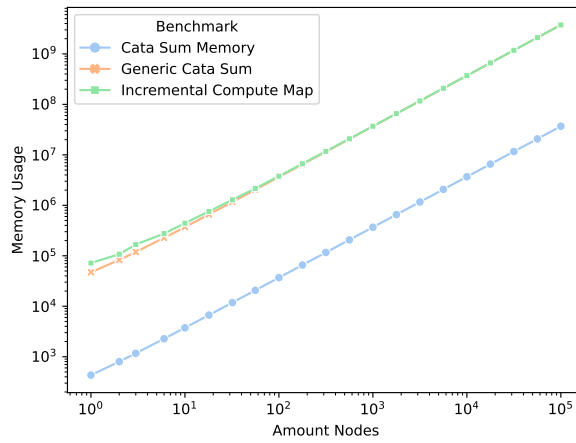


Figure 5.5: Overview memory usage

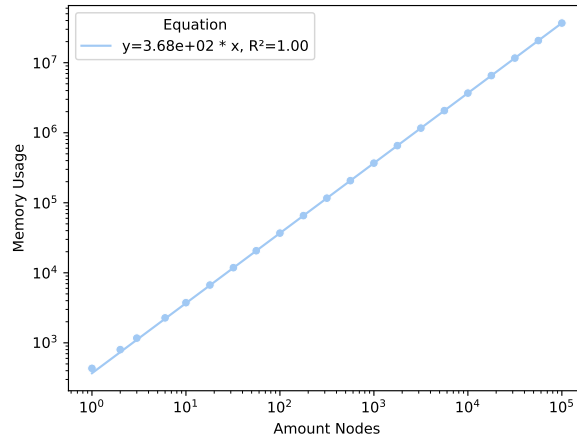


Figure 5.6: Memory usage for Cata Sum

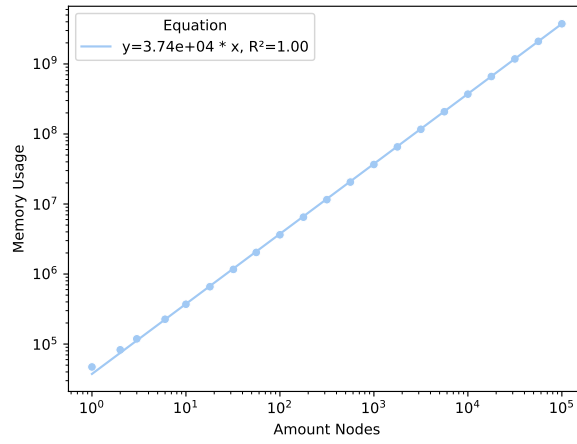


Figure 5.7: Memory usage for Generic Cata Sum

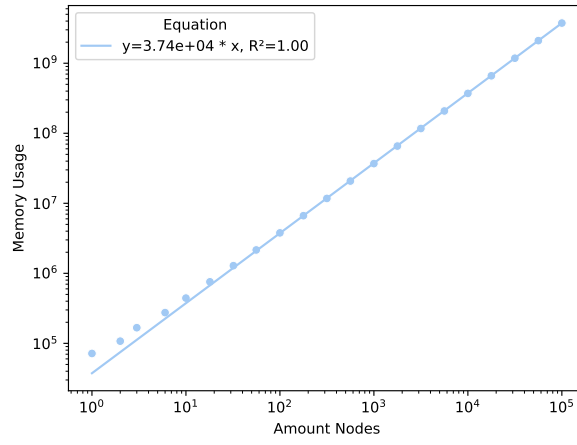


Figure 5.8: Memory usage for Incremental Cata Sum

5.3 Comparison Memory Strategies

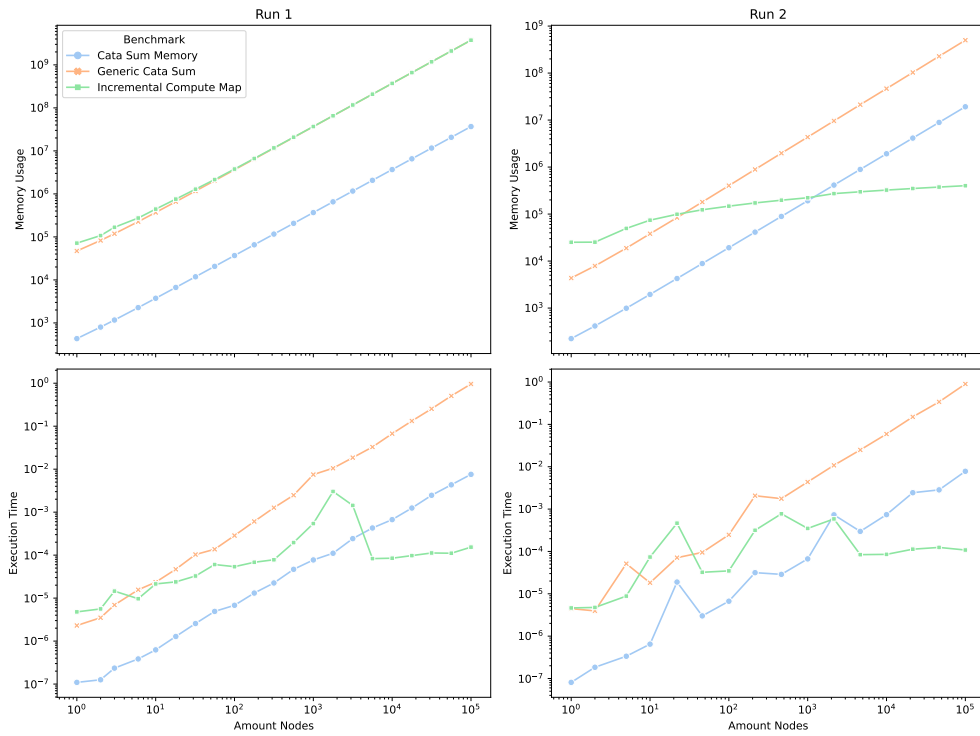


Figure 5.9: Comparison Memory Strategy

6

Conclusion and Future Work

6.1 Conclusion



Implementation Memo Cata

A.1 Definition Generic Datatypes

```
data U r      = U
data I r      = I r
data K a r    = K a
data (:+:) f g r = L (f r) | R (g r)
data (:*: ) f g r = (f r) :*: (g r)
data C c f r  = C (f r)

newtype Fix f = In { out :: f (Fix f) }
```

A.2 Implementation Hashable

```
class Hashable f where
  hash :: f (Fix (g :*: K Digest)) -> Digest

instance Hashable U where
  hash _ = digest "U"

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

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

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]

```

A.3 Implementation Merkle

```

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

merkleG :: Hashable f
  => f (Fix (g :: K Digest))
  -> (f :: K Digest) (Fix (g :: K Digest))
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

```

A.4 Implementation Cata Merkle

```

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

cataMerkle :: (Functor f, Traversable f)
  => (f a -> a) -> Fix (f :: K Digest) -> (a, M.Map Digest a)
cataMerkle alg t = runState (cataMerkleState alg t) M.empty

```

A.5 Implementation Zipper Merkle

```
data Loc :: * -> * where
  Loc :: (Zipper a) => Merkle a
      -> [Ctx (a :: K Digest) (Merkle a)]
      -> Loc (Merkle a)

modify :: (a -> a) -> Loc a -> Loc a
modify f (Loc x cs) = Loc (f x) cs

updateDigest :: Hashable a => Merkle a -> Merkle a
updateDigest (In (x :: _)) = In (merkleG x)

updateParents :: Hashable a => Loc (Merkle a) -> Loc (Merkle a)
updateParents (Loc x []) = Loc (updateDigest x) []
updateParents (Loc x cs) = updateParents
    $ expectJust "Exception: Cannot go up"
    $ up (Loc (updateDigest x) cs)

updateLoc :: Hashable a => (Merkle a -> Merkle a)
    -> Loc (Merkle a) -> Loc (Merkle a)
updateLoc f loc = if top loc'
    then loc'
    else updateParents
    $ expectJust "Exception: Cannot go up" (up loc')

where
  loc' = modify f loc
```


B

Regular

B.1 Zipper

```
data instance Ctx (K a) r
data instance Ctx U r
data instance Ctx (f :+: g) r = CL (Ctx f r) | CR (Ctx g r)
data instance Ctx (f :*: g) r = C1 (Ctx f r) (g r) | C2 (f r) (Ctx g r)
data instance Ctx I r = CId
data instance Ctx (C c f) r = CC (Ctx f r)
data instance Ctx (S s f) r = CS (Ctx f r)
```

Bibliography

- [1] Gérard Huet. “The zipper”. In: *Journal of functional programming* 7.5 (1997), pp. 549–554.
- [2] 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.