



**Utrecht
University**

Computing Science MSc Thesis

Incremental Computation for Algebraic Datatypes in Haskell

Author

Jort van Gorkum (6142834)

Supervisor

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
3	Prototype Implementation	6
3.1	Generic programming in Haskell	6
3.1.1	Pattern Functors vs Sums of products	6
3.1.2	Mutually recursive datatypes	6
3.1.3	Comparison Generic Libraries	6
3.2	Prototype language	6
3.3	HashMap vs Trie	8
4	Generic Implementation	9
4.0.1	Regular	9
4.1	Complexity	9
4.2	Memory Strategies	10
4.3	Pattern Synonyms	10
5	Experiments	11
5.1	Execution Time	11
5.2	Memory Usage	13
5.3	Comparison Memory Strategies	15
6	Conclusion and Future Work	16
6.1	Conclusion	16
A	Implementation Memo Cata	17
A.1	Definition Generic Datatypes	17
A.2	Implementation Hashable	17
A.3	Implementation Merkle	18
A.4	Implementation Cata Merkle	18
A.5	Implementation Zipper Merkle	19
B	Regular	20
B.1	Zipper	20

Todo list

Write abstract	1
Write a motivation	4
Write the contributions	4
Write the research questions	4
Write a piece about the use of Merkle Trees introduced by the Hdiff paper	5
Introduce generic programming and how it works	6
Describe the differences between defining generic data types	6
Describe what mutually recursive datatypes are and why do we need to know about it . .	6
Describe the use of the Zipper and how the hashes are updated	8
Write a piece about the comparison of storing it in a HashMap or a Trie datastructure . .	8
Write about Hdiff and the use of Trie datastructure	8
Write about why Regular is chosen	9
Write about the implementation of Regular and what had to change compared to the prototype language	9
Describe for every function used the complexity and what leads to the complete complexity	9
Describe multiple memory strategies for keeping memory usage and execution time low .	10
Write about paper selective memoization	10
Explain Pattern Synonyms	10

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

Write a piece about the use of Merkle Trees introduced by the Hdiff paper

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

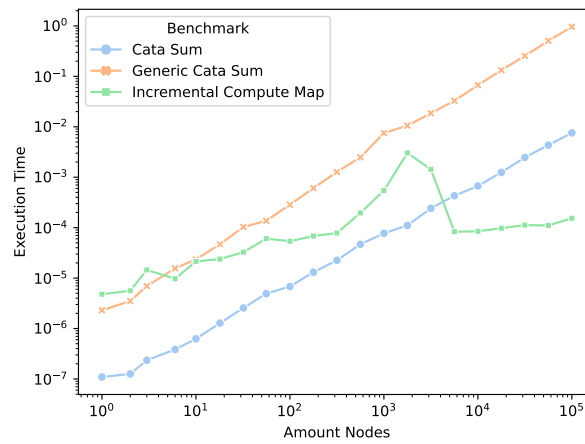


Figure 5.1: Overview execution time

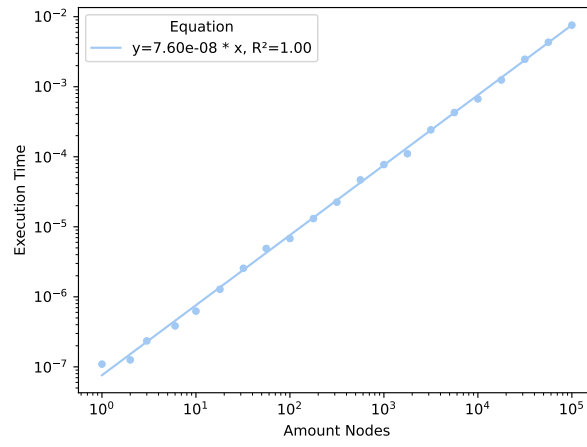


Figure 5.2: Execution time for Cata Sum

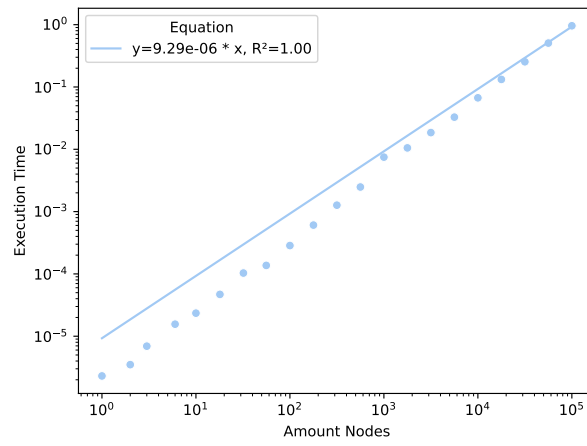


Figure 5.3: Execution time for Generic Cata Sum

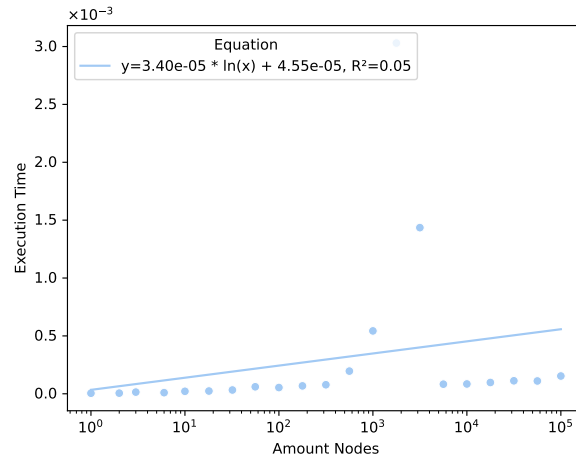


Figure 5.4: Execution time for Incremental Cata Sum

5.2 Memory Usage

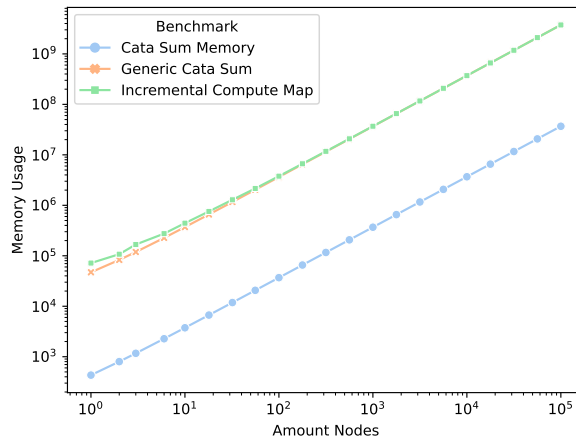


Figure 5.5: Overview memory usage

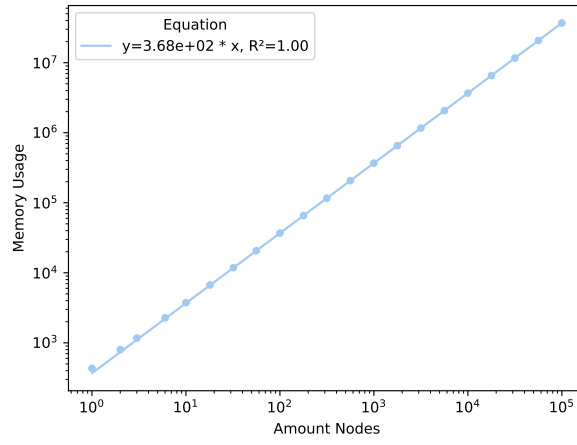


Figure 5.6: Memory usage for Cata Sum

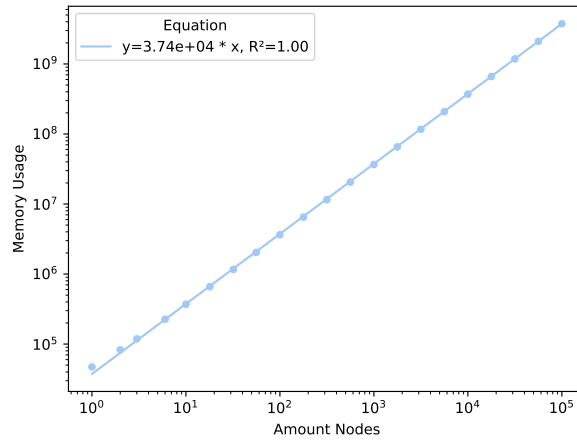


Figure 5.7: Memory usage for Generic Cata Sum

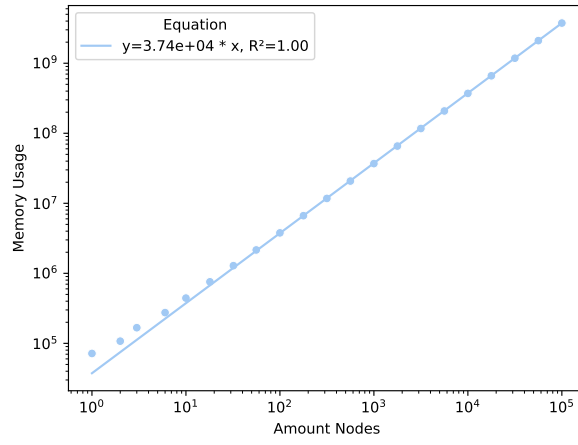


Figure 5.8: Memory usage for Incremental Cata Sum

5.3 Comparison Memory Strategies

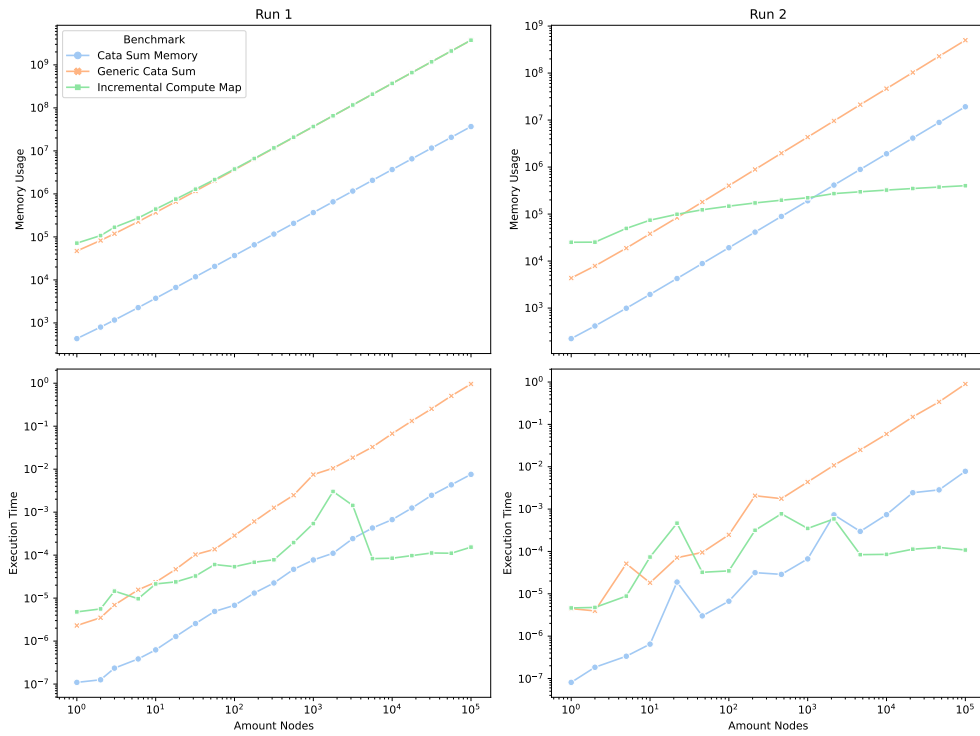


Figure 5.9: Comparison Memory Strategy

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)
```