Master's Thesis

# Incremental Cata Computation for Generic Data Types

Jort van Gorkum
Computing Science - Programming Technology

Supervisors:
Dr. Wouter Swierstra, Dr. Trevor McDonell

April 7, 2022

# 1 Abstract

# 2 Introduction

## 2.1 Problem statement

# 3 Background

## 3.1 Regular

Describe the workings of Regular with an example

## 3.2 HDiff

Write a piece about what HDiff does and how it suggests the use of Merkle trees

# 4 Implementation

Implementing the idea using a generic programming library, would be the ultimate goal. But first a **proof-of-concept** was made to show that the implementation is a viable product. A prototype-language is created, which is based on the notion of *pattern functors*.

## 4.1 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
```

## 4.2 Regular

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

# 5 Results

## 5.1 Memory

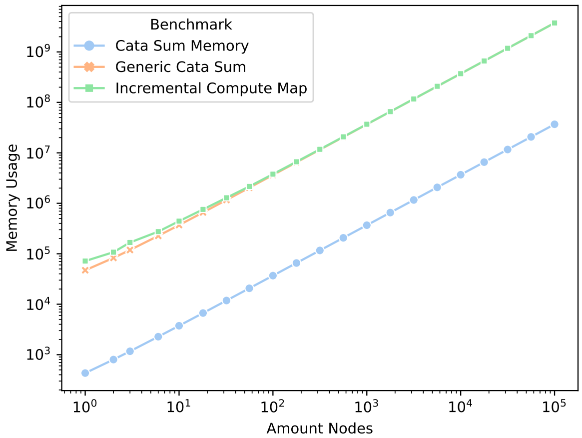


Figure 1: Overview memory usage

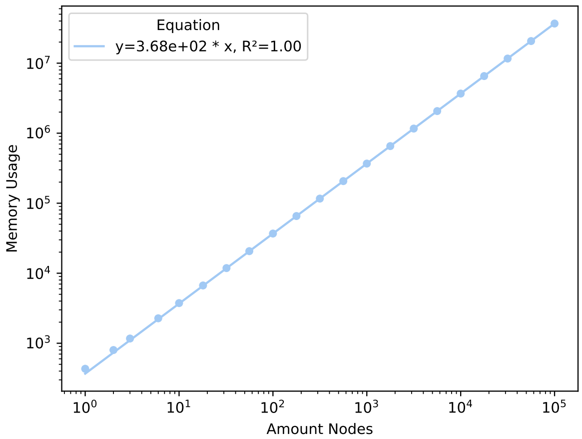


Figure 2: Memory usage for Cata Sum

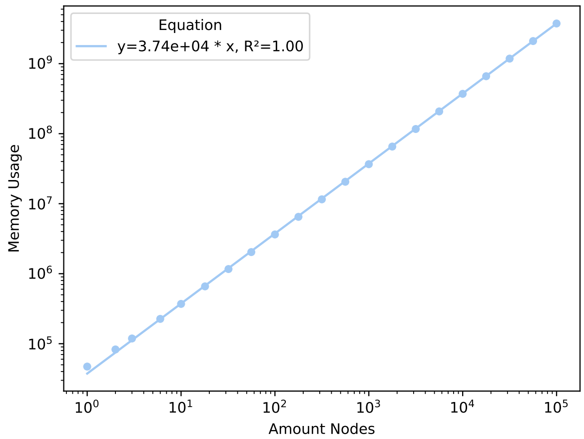


Figure 3: Memory usage for Generic Cata Sum

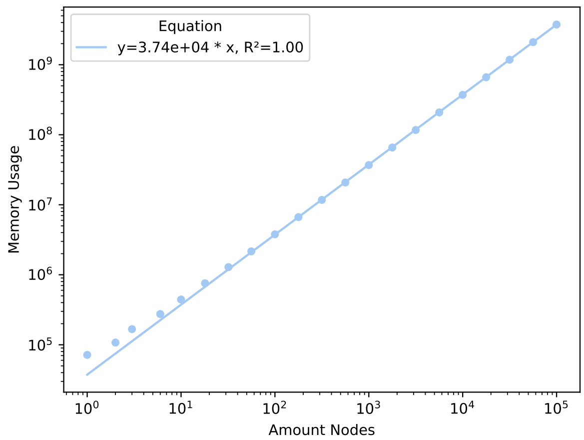


Figure 4: Memory usage for Incremental Cata Sum

# 6 Conclusion

# 7 Future Work

Can be used for efficiently updating the virtual DOM, for packages like, Elm & React

# 8 Appendix

# A Implementation Regular

## 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)
```

```haskell
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
```