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 29, 2022

Write abstract

Contents

Todo list

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

```haskell
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:

```haskell
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.

Add an example of hash collision probability for popular hash function

```
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
```

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

```haskell
            (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[4] 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 `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.

```haskell
data TreeH a = LeafH Hash a
             | NodeH Hash (Leaf a) a (Leaf a)
```

```haskell
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.

```haskell
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[2].

2.2 Zipper

The Zipper is a technique of representing a data structure by keeping track of how the data structure is being traversed through. The Zipper was first described by Huet[2] and is a solution for efficiently updating pure recursive data structures in a purely functional programming language (e.g., Haskell). This is accomplished by keeping track of the downward current subtree and the upward path, also known as the *location*.

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

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

type Loc a = (Tree a, Cxt a)

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

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

```
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)
```

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.

```
leave :: Loc a -> Loc a
```

```haskell
leave l@(t, Top) = l
leave l = top (up l)
```

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.

```haskell
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.

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

3

Datatype-Generic Programming

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

An example of reducing manual implementations for datatypes is the *deriving* mechanism in Haskell. The built-in classes of Haskell, such as `Show, Ord, Read`, can be derived for a large class of datatypes. However, deriving is not supported for custom classes. Therefore, we use *Datatype-Generic Programming*[1] 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 exploits the structure of datatypes to define functions by induction over the type structure. 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`[3]. 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.

```haskell
data U r         = U                  -- Empty constructors
data I r         = I r                -- Recursive call
data K a r       = K a                -- Constants
```

```haskell
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:

```haskell
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
```

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 calculate the length for every pattern functor `f`.

```haskell
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.

```haskell
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.

```haskell
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 (Map.fromList [("1", 1), ("2", 2), ("3", 3)])
    3
```

3.1.1 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) }

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.

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

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

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

3.1.2 Sums of Products

A different way of describing datatypes in a generic representation, besides pattern functors, are *Sums of Products*[6] (SOP). SOP is a generic representation with additional constrictions which more faithfully reflects the Haskell datatypes. The definition of the SOP is done using a `Code` of kind `[[*]]`. The outer list describes the sum and each inner list the products. The ` sign lifts the list to a type, instead of a value. The defined `Code` is then induced to a generic representation.

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

The usage of SOP makes it easier for developers to implement generic functionality. However, whereas pattern functors do not have structural constraints, SOP has, making it more complex to add extra functionality to SOP.

3.1.3 Mutually recursive datatypes

A large class of datatypes is supported by the previous section, namely *regular* types. Regular types are types in which the recursion only goes into the same datatype. However, if we want to support the syntax 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)
```

Describe what the challenges are with supporting mutually recursive datatypes

3.2 Comparison Generic Libraries

"Comparing libraries for generic programming in Haskell"[5]

Library	**Representation**	**Mutually Recursive**	**Complexity**
`regular`	Pattern Functor	$\times$	Low
`multirec`	Pattern Functor	$\checkmark$	Medium
`generics-sop`	SOP	$\times$	Medium-High
`generics-mrsop`	SOP	$\checkmark$	High

4 Prototype Implementation

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
```

```haskell
    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.

```haskell
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.

```haskell
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`.

```haskell
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
```

4.1.1 Zipper

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

4.2 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

5

Generic Implementation

5.1 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
  )
```

5.2 Complexity

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

5.3 Memory Strategies

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

Write about paper selective memoization

5.4 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
  )
```

6 Experiments

6.1 Execution Time

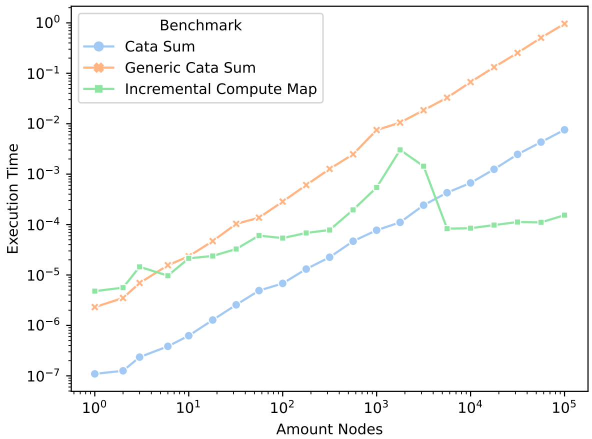

Figure 6.1: Overview execution time

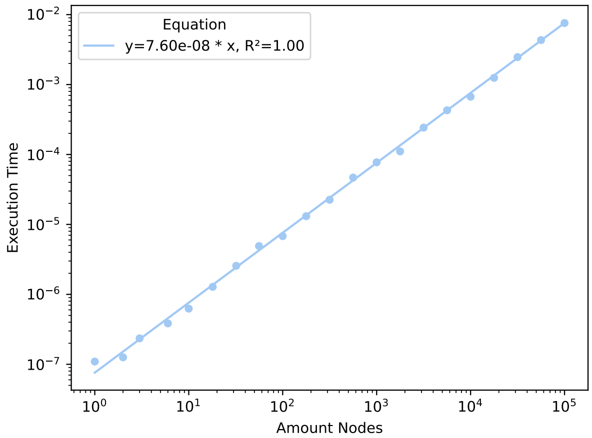

Figure 6.2: Execution time for Cata Sum

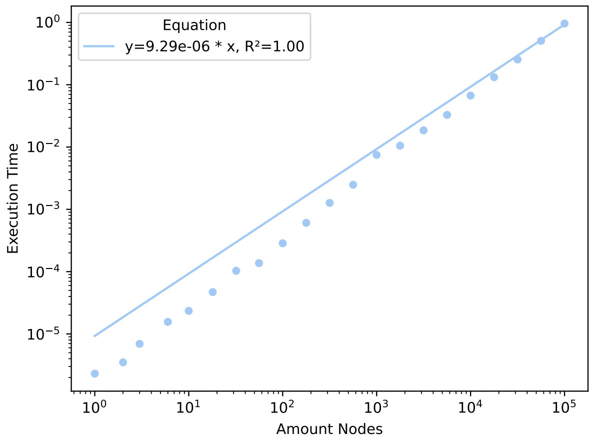

Figure 6.3: Execution time for Generic Cata Sum

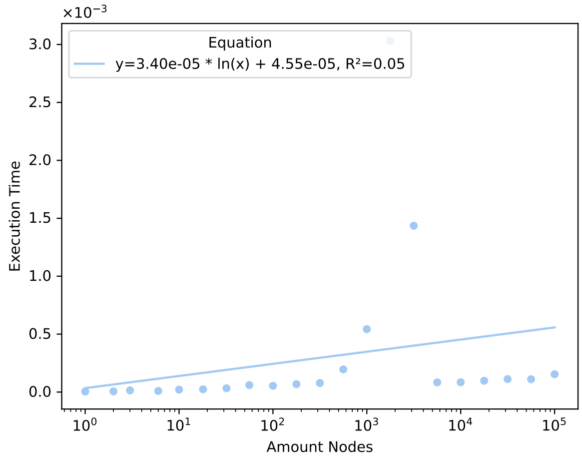

Figure 6.4: Execution time for Incremental Cata Sum

6.2 Memory Usage

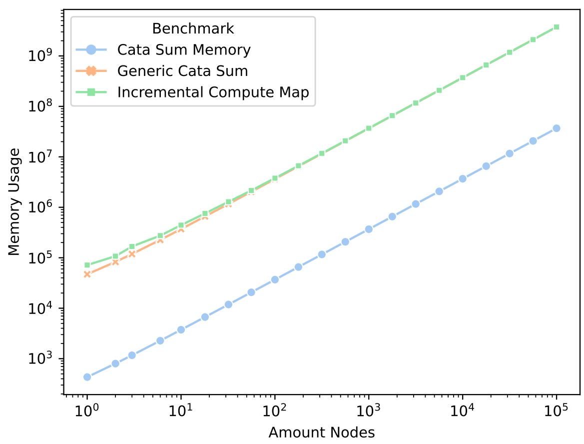

Figure 6.5: Overview memory usage

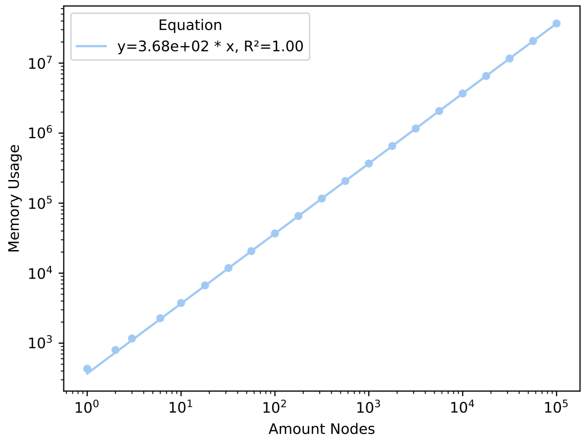

Figure 6.6: Memory usage for Cata Sum

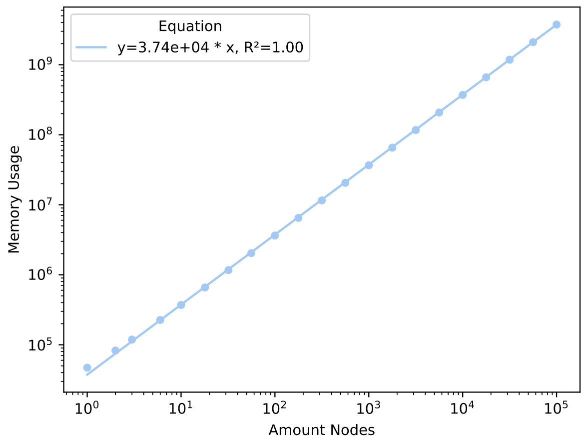

Figure 6.7: Memory usage for Generic Cata Sum

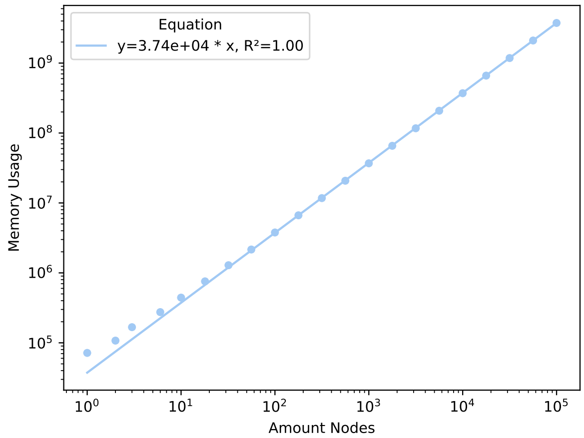

Figure 6.8: Memory usage for Incremental Cata Sum

6.3 Comparison Memory Strategies

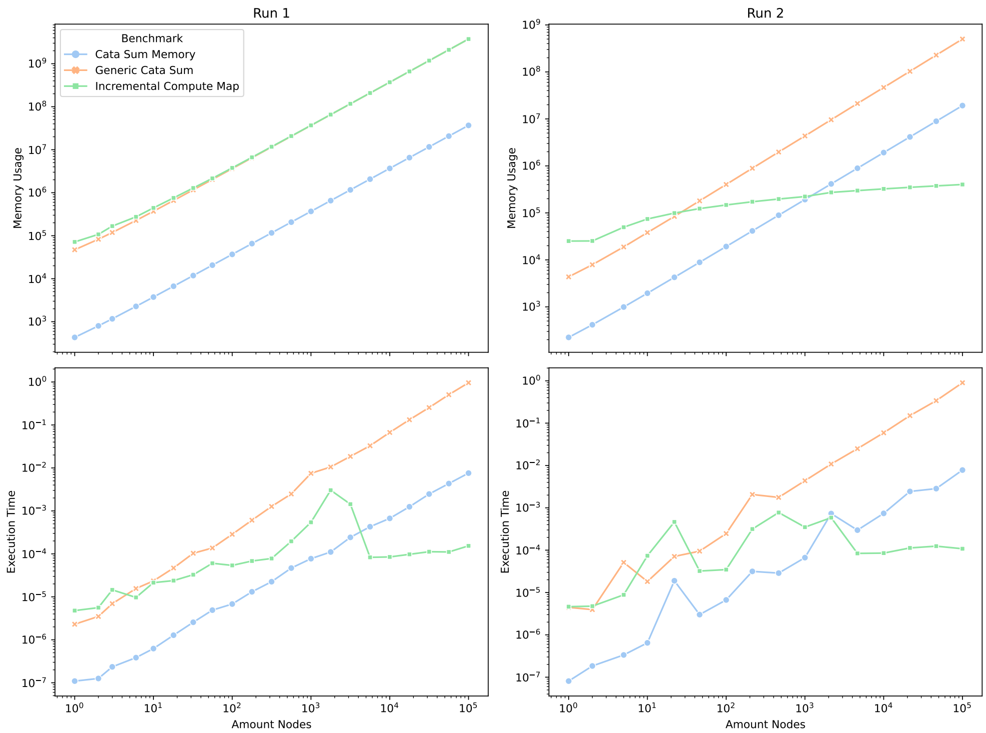

Figure 6.9: Comparison Memory Strategy

7

Conclusion and Future Work

7.1 Conclusion

A

Generic Programming

A.1 Instances Functor 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

Implementation Memo Cata

B.1 Implementation Hashable

```haskell
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]
```

B.2 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
```

B.3 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
```

B.4 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
```

```haskell
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
```

C

Regular

C.1 Zipper

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

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

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

[4] 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.

[5] Alexey Rodriguez et al. "Comparing libraries for generic programming in Haskell". In: *ACM Sigplan Notices* 44.2 (2008), pp. 111–122.

[6] 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.