Utrecht University

Computing Science MSc Thesis

Incremental Computation for Regular Datatypes in Haskell

Author
Jort van Gorkum (6142834)

Supervisors
Wouter Swierstra
Trevor McDonell

Faculty of Science
Department of Information and Computing Sciences
Programming Technology

May 24, 2022

Write abstract

Contents

Todo list

1

Introduction

1.1 Motivation

When computing a result over a data type, a small change would lead to completely recomputing the result. Parts of the recomputation give the same results as the previous computation. By keeping track of the intermediate results and reusing the results when the input is the same, fewer computations have to be performed.

An example of computing a result over a data type is computing the **maximum path sum** over a binary tree. Given a `BinTree`, starting at the root node, find a path from the root node to the leaves that leads to the maximum total.

The implementation of the computation is done in `Haskell`. And the definition of the binary tree and example tree is:

```haskell
data BinTree = Leaf Int
             | Node BinTree Int BinTree

exampleTree :: BinTree
exampleTree = Node (Node (Leaf 8) 7 (Leaf 9)) 3 (Node (Leaf 5) 4 (Leaf 2))
```

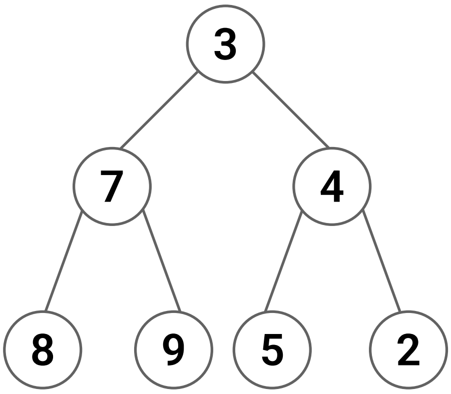


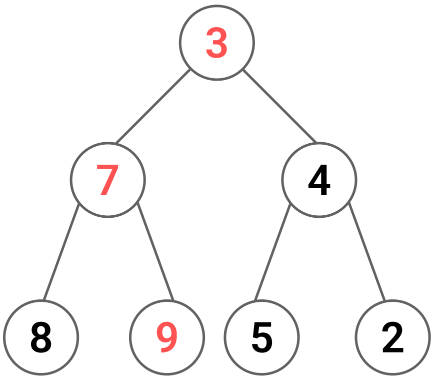
Figure 1.1: Graphic visualization of the `exampleTree`

An implementation of computing the max path sum over data type `BinTree` can been seen below and has a complexity of $\mathcal{O}(N)$, where N is the number of nodes in the tree.

```haskell
maxPathSum :: BinTree -> Int
maxPathSum (Leaf x)     = x
maxPathSum (Node l x r) = x + max (maxPathSum l, maxPathSum r)
```

And computing the max path sum over the `exampleTree` results in 19.

```haskell
maxPathSum exampleTree ≡ 19
```



To reduce the number of recomputations, first, we need to compare the structure of the data type for equality. When comparing the data type structure for equality, if they are equal, the previously computed result can be reused. Otherwise, the result needs to be recomputed. To compare the structure of the data type efficiently, we introduce the use of hash functions. Generating a hash every time a comparison takes place would be inefficient because part of the structure does not change, which leads to the same hash. Thus, every substructure in the data type stores the hash of its structure. Using hash functions and storing them to compare the structures of the data type for equality is performed in constant time ($\mathcal{O}(1)$).

In the example, a new data type is introduced: the `MerkleTree`. This data type is the same as the `BinTree`, but the constructors also contain a `Hash`. To create a `MerkleTree`, we traverse through a `BinTree` and hash the structure and store it. Creating a `MerkleTree` has a time complexity of $\mathcal{O}(N)$.

```haskell
data MerkleTree = LeafH Hash Int
                | NodeH Hash MerkleTree Int MerkleTree

merkle :: BinTree -> MerkleTree
merkle (Leaf x)     = LeafH (hash ["Leaf", x]) x
merkle (Node l x r) = NodeH (hash ["Node", x, hl, hr]) l' x r'
  where
    hl = getHash l'
    hr = getHash r'
    l' = merkle l
    r' = merkle r
```

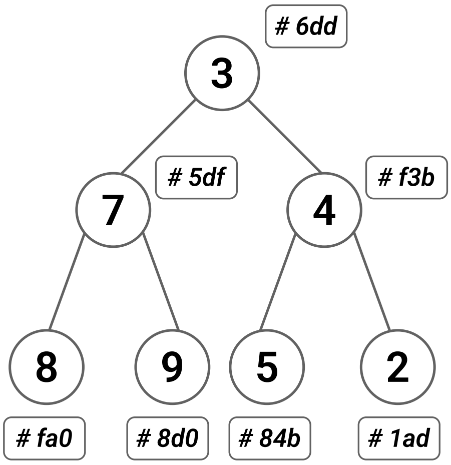
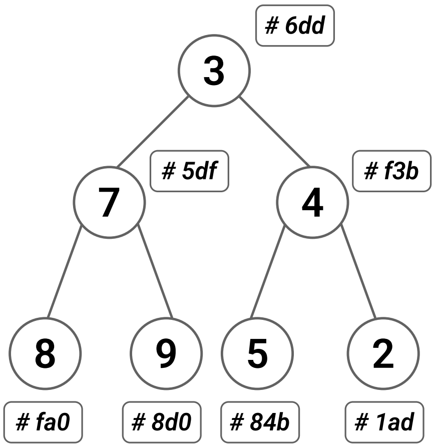


Figure 1.2: The `MerkleTree` of `exampleTree`

To compute the max path sum and its intermediate results, we can use the precomputed hashes of the `MerkleTree` to efficiently generate a `Map Hash Int`. The complexity of computing the max path sum and the intermediate results is $\mathcal{O}(N)$.

The example implementation of computing the max path sum and its intermediate results can be seen below. However this implementation uses a `union` (`<>`) to combine the `Map`'s which has a complexity of $\mathcal{O}(m * log(n/m + 1)), m <= n$[**HaskellDataUnion**]. This can be implemented more efficiently by giving the `Map` to the left side and then to the right side of the Tree, which makes it a constant operation. Except, this would make the code more complex. So, for clarity, the example is implemented with a union.

```
maxPathSumInc :: MerkleTree -> (Int, Map Hash Int)
maxPathSumInc (LeafH h x)     = (x, insert h x empty)
maxPathSumInc (NodeH h l x r) = (y, insert h y (ml <> mr))
  where
    y = x + max (xl, xr)
    (xl, ml) = maxPathSumInc l
    (xr, mr) = maxPathSumInc r
```



Hash Nodes	Max Sum
# 6dd	19
# 5df	16
# fa0	8
# 8d0	9
# f3b	9
# 84b	5
# 1ad	2

Figure 1.3: The `MerkleTree` with intermediate results

When there is a change in the `BinTree`, only the hashes of the change itself and its parents need to be recomputed. The recomputation of the hash has a time complexity with an upper bound of $\mathcal{O}(N)$ and an average of $\Theta(M \log N)$ where M is the number of changed nodes. The upper bound is $\mathcal{O}(N)$, because if $N = M \Rightarrow \mathcal{O}(N \log N) > \mathcal{O}(N)$, but when every node in the tree is changed, then the original function can be used. Meaning that this implementation only works efficiently with small changes to the data type.

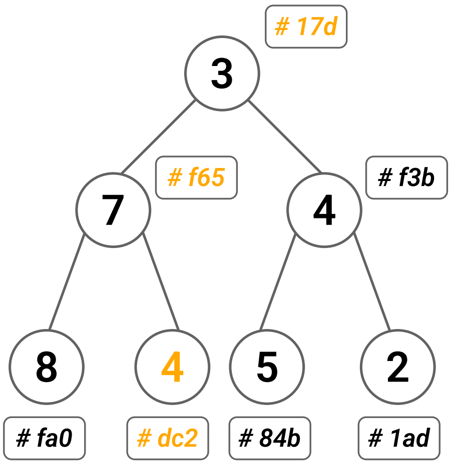


Figure 1.4: Changed Merkle Tree

Then to compute the max path sum over the Changed Tree, the previously computed `Map` can be used to reduce the amount of recomputation. This function also has the same complexity as the change Merkle tree, with an upper bound of $\mathcal{O}(N)$ and an average of $\Theta(M \log N)$.

```haskell
maxPathSumMap :: Map Hash Int -> MerkleTree -> (Int, Map Hash Int)
maxPathSumMap m (LeafH h x) = case lookup m h of
  Just y  -> (y, m)
  Nothing -> (x, insert h x m)
maxPathSumMap m (NodeH h l x r) = case lookup m h of
  Just y  -> (y, m)
  Nothing -> (y, insert h y (ml <> mr))
    where
      y = x + max (xl, xr)
      (xl, ml) = maxPathSumMap m l
      (xr, mr) = maxPathSumMap m r
```

In Table 1.1 and 1.2 the function complexities can be seen of the functions used to compute incrementally and complete. As can be seen, the upper bound complexities of the functions for incremental computation are the same as for the complete computation. However, when there are small changes to the data type the incremental update of the result has lower complexity ($\Theta(M \log N)$) than the complete recomputation complexity ($\Theta(N)$), which makes it more efficient.

Function	**Average**	**Upperbound**
`merkle`	$\Theta(N)$	$\mathcal{O}(N)$
`maxPathSumInc`	$\Theta(N)$	$\mathcal{O}(N)$
Change Merkle Tree	$\Theta(M \log N)$	$\mathcal{O}(N)$
`maxPathSumMap`	$\Theta(M \log N)$	$\mathcal{O}(N)$

Table 1.1: Functions used for incremental computation

Function	**Average**	**Upperbound**
`maxPathSum`	$\mathcal{O}(N)$	$\mathcal{O}(N)$

Table 1.2: Functions used for complete computation

The problem with this implementation is that it only works for the `BinTree` datatype. The goal would be to create a generic function, where only the `maxPathSum` would be defined and the result would automatically contain the intermediate results. A generic definition could look like this:

```haskell
cataMerkle :: (f a -> a) -> Map Hash a -> Fix (f :*: K Hash) -> (a, Map Hash a)
```

Using the `cataMerkle` function would lead to only needing to implement the `maxPathSum` function and the intermediate results are then automatically stored.

1.2 Research Questions

Write the research questions

1.3 Contributions

Write the contributions

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.

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. Using the formula $p = \epsilon^{\frac{-k(k-1)}{2N}}$ from *Hash Collision Probabilities*[10], we can calculate a 50% chance of getting a hash collision with a collection of k. The hash function CRC-32 needs a collection of 77163 hash values. The hash function MD5 needs a collection of 5.06×10^9 hash values. And, the hash function SHA-1 needs a collection of 1.42×10^{24} hash values. As a result of, we can say that for most popular hash functions, hash collisions are 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
```

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
      (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[7] 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.

```
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[5].

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[5] 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.

```haskell
leave :: Loc a -> Loc a
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)

> update (const (merkle (Leaf 4))) [down, left] (merkle (Node (Leaf 1) 2 (Leaf 3)))
```

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*[3] 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`[6]. 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) }

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*[11] (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.

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

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.

```
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*[12]. 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 `(Map Digest a) a`. The `cataMerkleState` function starts with retrieving the `State`, which keeps track of the intermediate results and stores them into a `Map Digest a`. Then, given the hash value of the recursive position, we look into the Map 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 `Map` and return the computed value.

```
cataMerkleState :: (Functor f, Traversable f)
                => (f a -> a) -> Merkle f -> State (Map 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 `Map Digest a`. To simplify the use of `cataMerkleState`, we define a function `cataMerkle`, which executes the `cataMerkleState` with an empty `Map` and returns the final computed result and the final state as the result.

```
cataMerkle :: (Functor f, Traversable f)
           => (f a -> a) -> Merkle f -> (a, Map 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, Map 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[1].

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

[1] "*Type families are to vanilla data types what type class methods are to regular functions.*"[4]

```
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 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 Complexity

Move this to the introduction?

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

4.4 HashMap vs Trie

Advanced data structures[2]

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

4.5 Memory Strategies

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

Write about paper selective memoization

4.6 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*[9].

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`.

```haskell
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[2].

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

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

5

Experiments

5.1 Execution Time

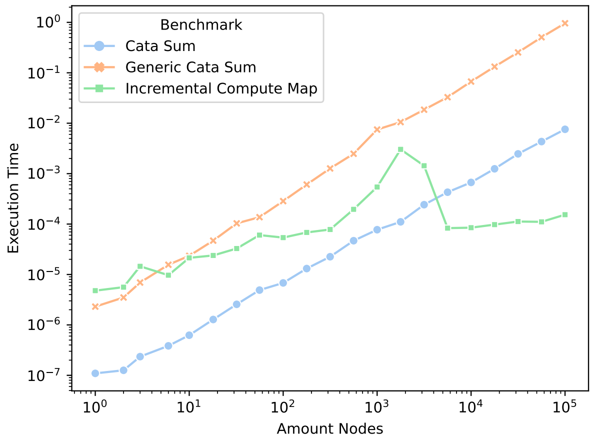


Figure 5.1: Overview execution time

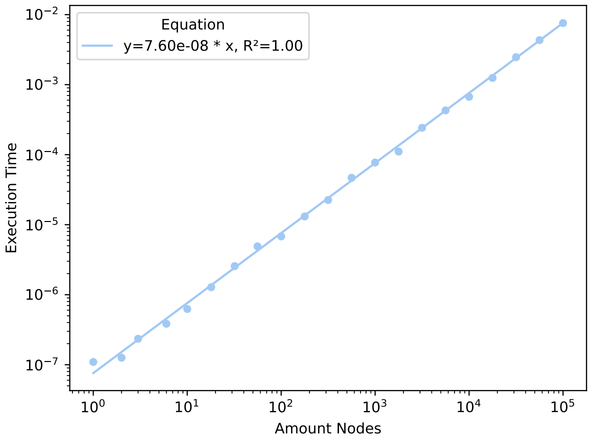


Figure 5.2: Execution time for Cata Sum

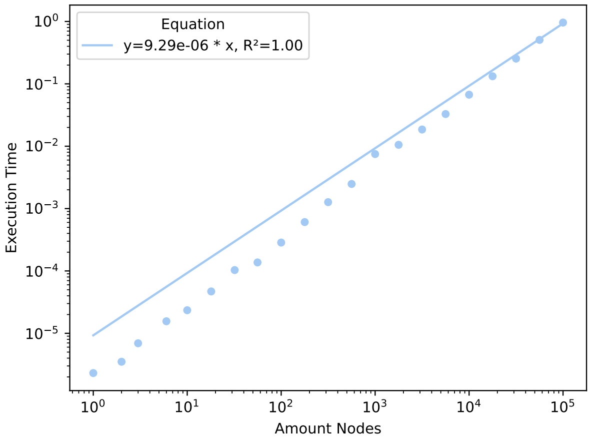


Figure 5.3: Execution time for Generic Cata Sum

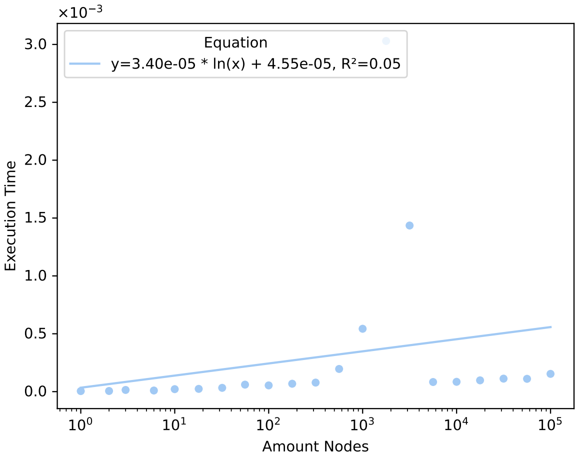


Figure 5.4: Execution time for Incremental Cata Sum

5.2 Memory Usage

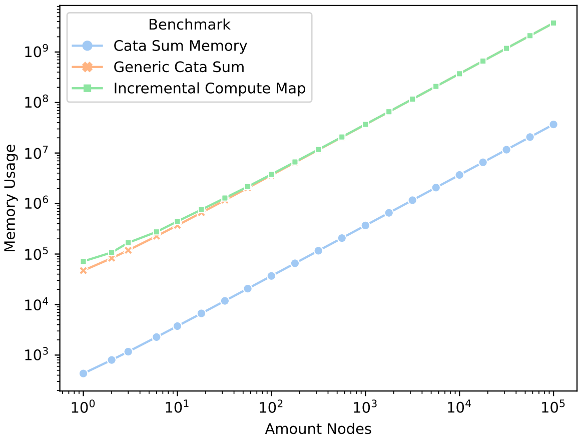


Figure 5.5: Overview memory usage

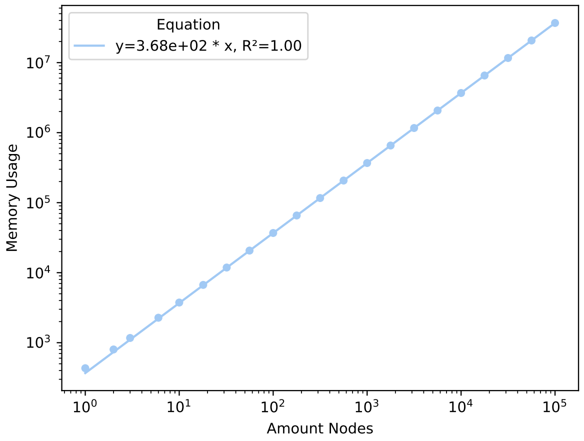


Figure 5.6: Memory usage for Cata Sum

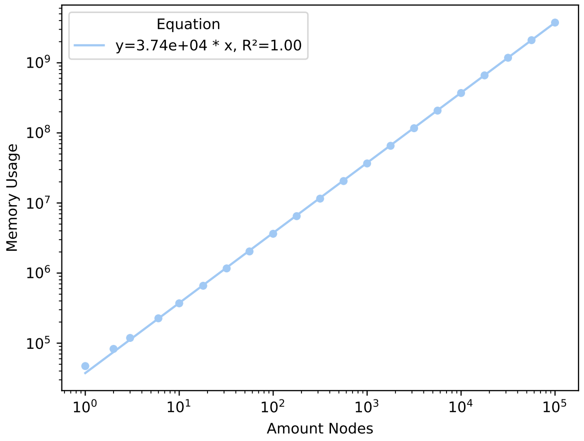


Figure 5.7: Memory usage for Generic Cata Sum

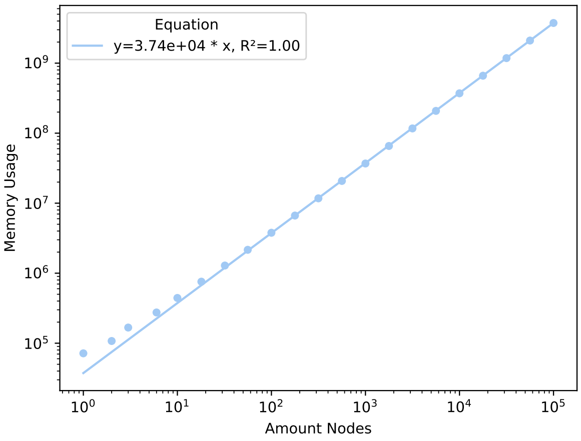


Figure 5.8: Memory usage for Incremental Cata Sum

5.3 Comparison Memory Strategies

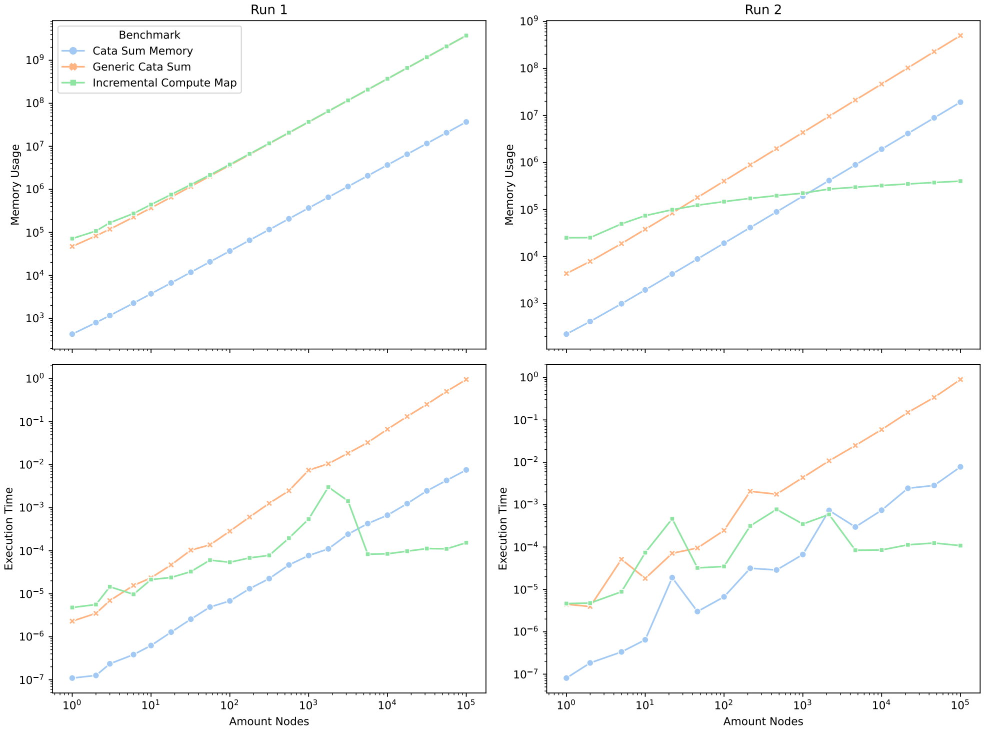


Figure 5.9: Comparison Memory Strategy

6
Conclusion and Future Work

6.1 Conclusion

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
```

Bibliography

[1] 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.

[2] Peter Brass. *Advanced data structures.* Vol. 193. 2008, pp. 336–356.

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

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

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

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

[7] 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.

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

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

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

[11] 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.

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