Utrecht University

# Generic Incremental Computation for Regular Datatypes

Jort van Gorkum

August 15, 2022

# Title Explanation – Incremental Computation

**Generic Incremental Computation for Regular Datatypes**

=

Incremental computation tries to improve performance, when the same function is called again, by only computing the output of the changed input compared to the previous computations

# Title Explanation – Example Incremental Computation

```haskell
fib :: Int -> Int
fib 0 = 0
fib 1 = 1
fib n = fib (n - 1) + fib (n - 2)
```

**Call Hierarchy**

```
                 fib(4)
               /        \
          fib(3)          fib(2)
          /   \           /   \
     fib(2)   fib(1)  fib(1)   fib(0)
     /   \
fib(1)  fib(0)
```

# Title Explanation – Example Incremental Computation

A technique for implementing incremental computations is:

*Memoization*

stores the result of a computation and returns the cached result when the same input occurs again.

# Title Explanation – Example Incremental Computation

**New Call Hierarchy**

```
          fib(4)
        /        \
    fib(3)        fib(2)
    /   \
 fib(2)  fib(1)
 /   \
fib(1) fib(0)
```

| Function Call | Result |
|---|---|
| fib(2) | 1 |
| fib(3) | 2 |
| fib(4) | 3 |

**Cached Results**

# Title Explanation – Generic

**Generic Incremental Computation for Regular Datatypes**

=

Generic refers to *datatype-generic programming*, which is a form of abstraction that allows defining functions that can operate on a large class of datatypes.

# Title Explanation – Generic Example

```haskell
data List a = Nil | Cons a (List a) -- Haskell Notation [] | x : []

length :: List a -> Int
length Nil          = 0
length (Cons _ t) = 1 + length t

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

length :: Tree a -> Int
length Leaf _         = 1
length (Node l _ r) = 1 + length l + length r
```

# Title Explanation – Generic Example

```
gLength :: (Generic f) => f a -> Int
gLength = ...
```

A *single* `length` function can be written, that can operate on lists, trees, and many other datatypes

```
> gLength (Cons 1 (Cons 2 (Cons 3 Nil))) -- List Int
  3
```

```
> gLength (Node Leaf 1 Leaf) -- Tree Int
  3
```

# Title Explanation – Regular Datatypes

**Generic Incremental Computation for Regular Datatypes**

=

Regular datatypes are recursive datatypes, which can only recurse into themselves, such as lists, binary trees, etc.

# Title Explanation – Regular Datatypes Example

**Regular Datatypes**

```
data List a = Nil | Cons a (List a)

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

**Not Regular Datatypes**

```
data Tree a = Empty | Node a (Forest a)

data Forest a = Nil | Cons (Tree a) (Forest a)
```

# Title Explanation – Summary

**Generic Incremental Computation for Regular Datatypes**

- Improve performance by only recomputing changed input
- Using generic programming to define functionality for a large class of datatypes
- The algorithm works for the regular datatypes

# Goal

Implement an incremental algorithm which performs better than the non-incremental algorithm for large regular datatypes

# Solution – Example with Memoization

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

sumTree :: Tree Int -> Int
sumTree (Leaf x)     = x
sumTree (Node l x r) = x + sumTree l + sumTree r

exampleTree = Node (Node (Leaf 1) 3 (Leaf 2)) 5 (Node (Leaf 1) 3 (Leaf 2))
```

**Visual representation**

```
                5
              /   \
sumTree(    3       3     ) = 17
           / \     / \
          1   2   1   2
```

# Solution – Example with Memoization

**Memoized** version of the `sumTree`

### Example Tree

```
      5
    /   \
   3     3
  / \   / \
 1   2 1   2
```

| Tree | Result |
|---|---|
| 5<br>/ \\<br>3 3<br>/ \\ / \\<br>1 2 1 2 | 17 |
| 3<br>/ \\<br>1 2 | 6 |

**Cached Results**

# Solution – Equality

For every function call, the given tree needs to compare if there is an already cached input. This is accomplished by traversing through both trees.

```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
```

This is inefficient, and to improve the performance, we are going to use a *hash function*.

# Solution – Using Hash function

A *hash function* is a process of transforming input into an arbitrary fixed-size value (i.e., digest), where the same input always generates the same output.

```
       5
      / \
     3   3       ->    1da16c7c48e429b4ad6e6c88d941d2bd
    / \ / \
   1  2 1  2                         Digest
```

The comparison for equality is now **constant** time instead of *linear*.

**However**, the digest still needs to be computed for every comparison, while the structure stays the same. So, the digest needs to be stored somewhere.

# Solution – Storing the Digests

A *Merkle Tree* is a data structure which integrates the *digests*, which represents the internal structure, within the data structure.

```
data TreeH a = LeafH Digest a
             | NodeH Digest (TreeH a) a (TreeH a)
```

```
instance Eq (TreeH a) where
  LeafH d1 _     == LeafH d2 _     = d1 == d2
  NodeH d1 _ _ _ == NodeH d2 _ _ _ = d1 == d2
  _              == _              = False
```

# Solution – Efficiently updating the Input

When a new tree is given to the function, the digests of the entire tree needs to be rehashed. This is **inefficient**, because, the affected digests are the changed parts of the tree and their parent nodes.

A more **efficient** method is, by passing a set of instructions on how to update the current input into the newly changed input. Using the set of instructions we can:

- traverse through the data structure
- modify the value of the current position with a given function
- and to return to a complete tree, we traverse back to the rootnode, while updating the digests

To implement the previously described functionality, we use a *Zipper*.

# Solution – Efficiently updating the Input

```
     5                            5
   /   \                           \
  3     3   - Left ->    3          3    - Modify ->
 / \   / \              / \        / \
1   2 1   2            1   2      1   2
```

```
          5                      5
           \                   /   \
  7         3   - Up ->       7     3
 / \       / \               / \   / \
4   3     1   2             4   3 1   2
```

**Primitive Type Constructors**

```
data U r         = U                     -- Empty constructor
data I r         = I r                   -- Recursive position
data K a r       = K a                   -- Constant
data (f :+: g) r = L (f r) | R (g r) -- Sums (Choice)
data (f :*: g) r = (f r) :*: (g r)     -- Products (Combine)
```

**Pattern functor for the** `Tree` **datatype**

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

type instance PF (Tree a) = K a                      -- Leaf
                          :+: (I :*: K a :*: I) -- Node
```

# Generic Implementation – Hashing Primitive Type Constructors

```
class Hashable f where
  hash :: f (Merkle g) -> Digest
```

- The `f` represents the primitive type constructors (i.e., `U`, `I`, `K`, `:+:`, `:*:`)
- The `Merkle g` is the type of the recursive position (i.e., `I r`).
- `Merkle g` contains the `Digest` of its internal structure.
- The `hash` function only converts a single layer of the pattern functor.

# Generic Implementation – Hashing Primitive Type Constructors

```haskell
instance Hashable U where
  hash _ = hash "U"

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

# Generic Implementation – Hashing Primitive Type Constructors

```haskell
instance (Hashable f, Hashable g) => Hashable (f :+: g) where
  hash (L x) = digestConcat [hash "L", hash x]
  hash (R x) = digestConcat [hash "R", hash x]

instance (Hashable f, Hashable g) => Hashable (f :*: g) where
  hash (x :*: y) = digestConcat [hash "P", hash x, hash y]
```

# Generic Implementation – Hashing Primitive Type Constructors

```
class Hashable f where
  hash :: f (Merkle g) -> Digest
```

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

# Generic Implementation – Generic Merkle Tree

```haskell
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

> let x = merkle (Node (Leaf 1) 2 (Leaf 3))
    x :: (Merkle (PF (Tree Int)))
    x = In ((R (I (In (K 1)) :*: K 2 :*: I (In (K 3)))) -- PF (Tree Int)
              :*: K 1da16c7c48e)                         -- Merkle
```

# Generic Implementation – Cata Merkle

cata means *catamorphism* which is a generalization of a *fold*. A fold combines the data structure into a single value (e.g., `sumTree` is a fold).

```
cataMerkleState :: (Functor f, Traversable f)
                => (f a -> a) -> Merkle f -> State (HashMap Digest a) a
cataMerkleState alg (In (x :*: K d))
  = do m <- get
       case lookup d m of
         Just a  -> return a
         Nothing -> do y <- mapM (cataMerkleState alg) x
                       let r = alg y
                       modify (insert d r) >> return r
```

# Generic Implementation – Incremental Sum Tree

```haskell
incSumTree :: Merkle (PF (Tree Int)) -> (Int, HashMap Digest Int)
incSumTree = cataMerkle
  (\case
    L (K x)                 -> x
    R (I l :*: K x :*: I r) -> l + x + r
  )
```

```
> incSumTree (merkle (Node (Leaf 1) 2 (Leaf 3)))
    (6, {"931090e5": 1, "7d1ef1c9": 3, "ba811ed5": 6})
```

# Generic Implementation – Pattern Synonyms

*Pattern synonyms* add an abstraction over patterns, which allows the user to move additional logic from `guards` and `case` expressions into patterns.

```
incSumTree :: Merkle (PF (Tree Int)) -> (Int, HashMap Digest Int)
incSumTree = cataMerkle
  (\case
    Leaf_ x     -> x
    Node_ l x r -> l + x + r
  )
```

# Generic Implementation – Final Process

```
> let exampleTree = merkle (Node (Leaf 1) 2 (Leaf 3))

> let (y, m) = incSumTree exampleTree
    (6, {"931090e5": 1, "7d1ef1c9": 3, "ba811ed5": 6})

> let newTree = update [Left] (const (Leaf 4)) exampleTree

> incSumTreeMap m exampleTree
    (9, {"931090e5": 1, "7d1ef1c9": 3, "ba811ed5": 6, "61f159e6": 4
        , "16d55294": 9})
```

# Experiments – Method

**Three functions:**

- `Cata Sum`
  - Non-incremental algorithm (which computes sumTree)
- `Generic Cata Sum`
  - Incremental algorithm with an empty cache
- `Incremental Cata Sum`
  - Incremental algorithm with a filled cache

**Three scenarios:**

- Worst case: updates the lowest left leaf with a new leaf
- Average case: updates a node in the middle of the data structure with a new leaf
- Best case: updates the left child of the root-node with a new leaf

# Experiments – Results - Execution Time

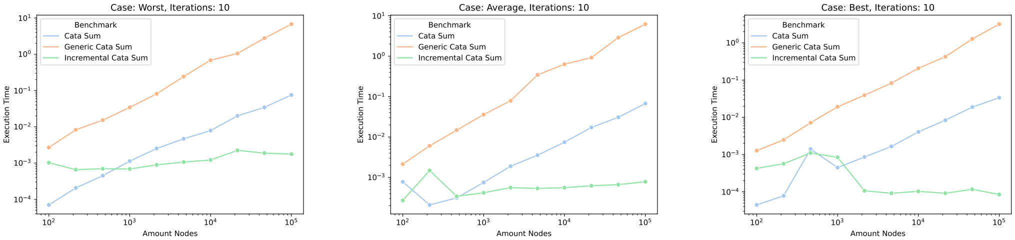


Figure: The execution time over 10 executions for the Worst, Average and Best case.

# Experiments – Results - Memory Usage

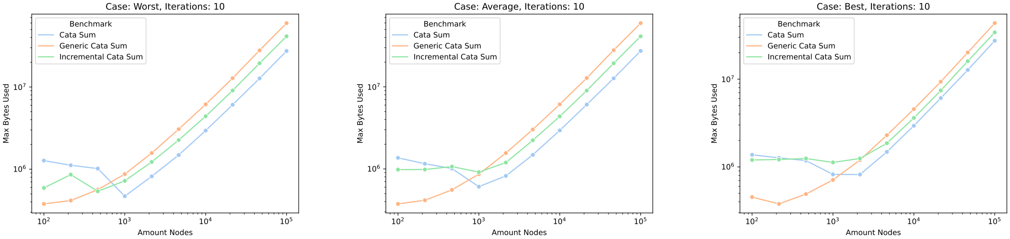


Figure: The max-bytes-used over 10 executions for the Worst, Average and Best case.

# Conclusion – Limitation & Future Work

**Limitations**

- Only supports Regular Datatypes
- Pattern Synonyms needs to be handwritten

**Future Work**

- Support Mutually Recursive Datatypes
- Generate Pattern Synonyms using `TemplateHaskell`

# Conclusion – Undiscussed Topics

- Cache management
- The explanation of fixed-point
- Implementation of the (Generic) Zipper

# Conclusion – Summary

- We have implemented an efficient incremental algorithm for regular datatypes
- The incremental algorithm is faster than the non-incremental version when the data structure contains more than $10^3$ nodes
- We use pattern synonyms, so that the developer experience is almost at the same level as the non-incremental implementation
- However, the initial pass of the incremental algorithm is a lot slower than the non-incremental version. Therefore, the incremental algorithm needs to be performed a lot (preferably, with small changes), before being overall faster than the non-incremental version.