

Generic Incremental Computation for Regular Datatypes

Jort van Gorkum

August 13, 2022

Generic **Incremental Computation** for Regular Datatypes

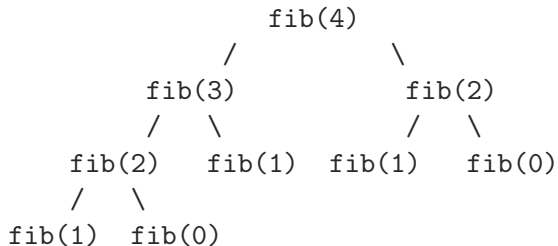
=

Incremental computation is an approach to improve performance by only recomputing result for changed input

Title Explanation – Example Incremental Computation

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

Call Hierarchy

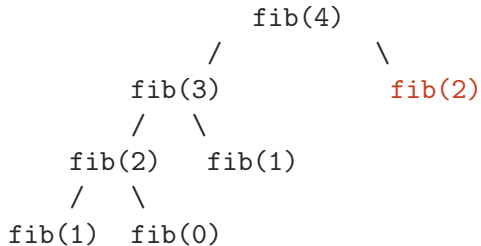


Memoization

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



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

Cached Results

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

```
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
```


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)
```

Generic Incremental Computation for Regular Datatypes

Goal – What does this solve?

Goal: implement an incremental algorithm which performs better than the non-incremental algorithm for large regular datatypes

Goal – Example Problem

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

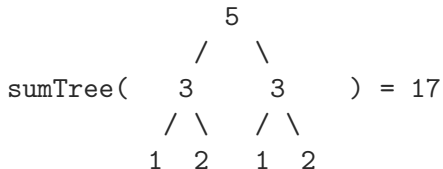
```
sumTree :: Tree Int -> Int
```

```
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))
```

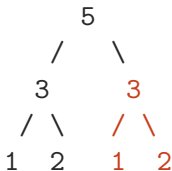
Visual representation

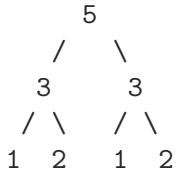



Goal – Example Problem

Memoized version of the sumTree

Example Tree

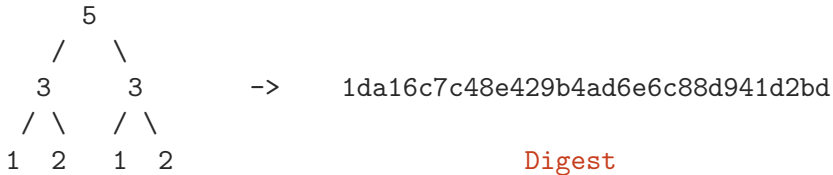


Tree	Result
	17
	6

Cached Results

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



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)
```

```
merkle :: Tree Int -> TreeH Int
merkle l@(Leaf x)      = LeafH (hash l) x
merkle b@(Node l x r) = NodeH (hash b) l' x r'
  where
    l' = merkle l
    r' = merkle r
```


Solution – Using Digests

```
sumTreeInc m (LeafH d x) = case lookup d m of  
  Just z  -> (z, m)  
  Nothing -> (x, insert d x m)
```

...

SumTreeInc

Solution – Updating the Digests

The *Zipper* is a technique for keeping track of how the data structure is being traversed through

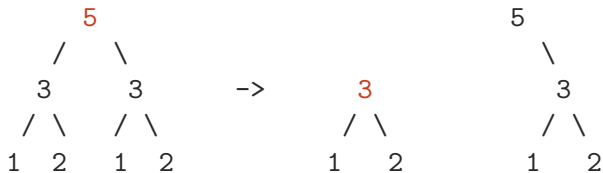
```
data Cxt a = L (Tree a) a | R (Tree a) a
```

```
type Loc a = (Tree a, [Cxt a])
```

```
enter :: Tree a -> Loc a
```

```
enter t = (t, [])
```

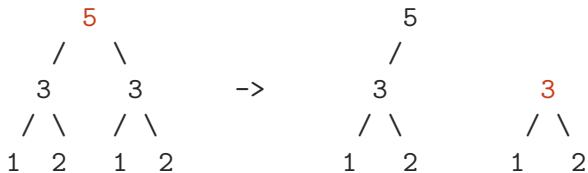
Solution – Zipper - Left



```
left :: Loc a -> Loc a
```

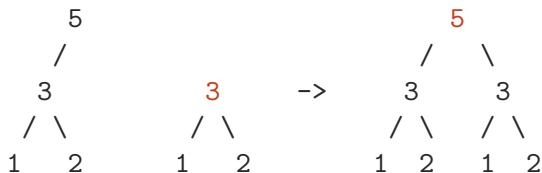
```
left (Node l x r, cs) = (l, (L r x):cs)
```

Solution – Zipper - Right



```
right :: Loc a -> Loc a
right (Node l x r, cs) = (r, (R l x):cs)
```

Solution – Zipper - Up



```
up :: Loc a -> Loc a
up (t, (L r x):cs) = (Node l x r, cs)
up (t, (R l x):cs) = (Node l x r, cs)
```

Generic Programming – Pattern Functors

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

```
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

```
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 [digest "I", getDigest x]
  where
    getDigest :: Fix (f :: K Digest) -> Digest
    getDigest (In (_ :: K h)) = h
```

Generic Implementation – Generic Merkle Tree

```
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
```

Generic Implementation – Cata Merkle

```
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 – Cata Sum

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

> cataSum (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.

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

Experiments – Method

Three functions:

- Cata Sum
 - ▶ Non-incremental algorithm
- Generic Cata Sum
 - ▶ Incremental algorithm with an empty cache
- Incremental Cata Sum
 - ▶ Incremental algorithm with a filled cache

Three scenarios:

- Worst case: ...
- Average case: ...
- Best case: ...

I am fast as f*ck boy!

Conclusion – Summary

- ...