Generic Incremental Computation for Regular Datatypes

Jort van Gorkum

August 12, 2022

Title Explanation – Incremental Computation

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

```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

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)))
    2
```

```
> gLength (Node Leaf 1 Leaf)
    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, Nodeary 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)
```

Problem statement – What is the problem?

- Memoization is dependent on the size of the input
- Becomes a problem with large recursive data structures (e.g., a Tree)

Problem statement – Example

```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
```

Problem statement – Example

Incremental Compute `sumTree`

Example Tree

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

<table>
<tr><th>Tree</th><th>Result</th></tr>
<tr><td>5
/ \
3 3
/ \ / \
1 2 1 2</td><td>17</td></tr>
<tr><td>3
/ \
1 2</td><td>6</td></tr>
</table>

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

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

Solution – Implementation Hash function

```haskell
class Hashable a where
  hash :: a -> Digest

instance Hashable a => Hashable (Tree a) where
  hash (Leaf x)     = concatDigest [hash "Leaf", hash x]
  hash (Node l x r) = concatDigest [hash "Node", hash l, hash x, hash r]
```

Solution – Storing the Digests

A *Merkle Tree* is a data structure which integrates the *digests* 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

`todo`

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 Programming – Fixed-point

Pattern functors only represent a single layer of recursion

```haskell
data Fix f = In (f (Fix f))

type Tree a = Fix (PF (Tree a))
```

Isomorphic

```haskell
from :: Tree a -> Fix (PF (Tree a))
from (Leaf x)       = In (K x)
from (Node l x r) = In (I (from l) :*: K x :*: I (from r))

to :: Fix (PF (Tree a)) -> Tree a
to (In (K x))                       = Leaf x
to (In (I l :*: K x :*: I r)) = Node (to l) x (to r)
```

Generic Implementation – Merkle

```haskell
type Merkle f = Fix (f :*: K Digest)

type MerkleTree a = Merkle (PF (Tree a))

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

instance ...

merkle ...
```

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: ...

Experiments – Results

I am fast as f*ck boy!

Conclusion – Future work - Pattern Synonyms

- ...

Conclusion – Future work - Support Mutually Recursive Datatypes

- ...

Conclusion – Summary

- …