Generic Incremental Computation for Regular Datatypes

Jort van Gorkum

August 11, 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)
```

Memoization

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 | Bin (Tree a) a (Tree a)

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

Title Explanation – Generic Example

```haskell
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 (Bin 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, binary trees, etc.

Title Explanation – Regular Datatypes Example

Regular Datatypes

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

data Tree a = Leaf | Bin 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?

Problem statement – Example

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

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

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

Visual representation

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

Problem statement – Example

Example Tree

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

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

Cached Results