Non-negative sum monoid solved in Haskell

Denis Grotsev
denis.grotsev@gmail.com

August 23, 2021

Abstract

Non-negative balance constraint for sum of changes is so ubiquitous real world task and so easy to solve that it is hard to find out the problem: incremental and MapReduce-style computing is not applicable due to lack of associativity.

The paper constructs monoid structure using the programming language Haskell to prepare the algorithm for incremental, parallel and distributed computing.

Keywords: monoid, non-negative sum, grosum, MapReduce, incremental computing, parallel computing, distributed computing

1 Introduction

Material world quantities are naturally expressed by non-negative numbers: remaining goods in stock or cash in a wallet. Given a list of changes it is possible to specify non-negative sum where negative part of balance is discarded.

```
spec :: [Integer] ->  Integer
spec changes = foldl' (⊞) 0  changes
```

Rectified plus binary operation is defined as positive part of usual plus.

```
x ⊞ y = max 0 (x + y)
```

Unfortunately, rectified plus is not associative. The paper constructs associative monoid structure to wrap intermediate calculations of non-negative sum. This opens up the opportunity of incremental, parallel and distributed computing.

2 Monoid instance

Non-negative sum calculation intermediate state is similar to Grothendieck group construction and consists of negative fall part and positive growth part of balance.

```
data Gro n = Gro { fall :: n , grow :: n }
```

Lifting a numerical change into monoid structure splits this change into positive and negative parts.

```
lift c = if c < 0 then Gro (-c) 0 else Gro 0 c
```

Semigroup instance merges two adjacent intermediate states together. Firstly, previous growth compensates following fall. Secondly, lifted compensation component-wise adds up to the remaining previous fall and following growth.

```
instance (Num n, Ord n) => Semigroup (Gro n) where
    a <> b = Gro (fall a + fall c)
                 (grow b + grow c)
      where
        c = lift (grow a - fall b)
```

Empty element of monoid has no fall and no growth.

```
instance (Num n, Ord n) => Monoid (Gro n) where
    mempty = Gro 0 0
```

Non-negative sum just extracts growth from intermediate state and forgets fall.

```
grosum :: (Num n, Ord n) => [n] -> n
grosum = grow . foldMap lift
```

3 Proof

Quickcheck tests that specification is satisfied by solution and solution demonstrates monoid properties.

```
prop_eq xs = spec xs == grosum xs
prop_monoid = monoid (mempty :: Gro Integer)
```

Associativity is formally provable by substitution and based on properties of positive and negative parts.

```
max 0 (x + max 0 y) == max 0 x + max 0 (y + min 0 x)
min 0 (x + min 0 y) == min 0 x + min 0 (y + max 0 x)
```

These equations are trivially checked for each of the four quadrants of (x, y) plane.