INCREMENTAL COMPUTATION WITH DIVIDE AND CONQUER MEMOIZATION

Robbert van Dalen
robbert.van.dalen@gmail.com

Abstract

There are many ways to implement incremental computation. One popular method - implemented by spreadsheets - is to keep an up-to-date topologically sorted dependency graph of all cells.
After modification of a cell, a spreadsheet can optimally determine which cells need to be recalculated (without recalculating everything) given its dependency graph.
In this short paper we will discuss another approach that is based on memoization.
An example is given that uses a divide and conquer algorithm to maximally re-use sums of integers. Next to that, we sketch a possible implementation based on uniquely represented confluently persistent treaps.

A discrete function

We start our discussion with the definition of a simple **discrete** function `f` which maps a set of integers `1..8` to another set of multiples: `2,4..14,16`.

```
f(x: Int): Int = 2*x                  x = 1..8
```

Memoization

We use the above definition `f` to derive a binary relation `fR`. We do not built relation `fR` upfront as we can do that lazily while evaluating `f`. This is called **memoization**.

fR	
1	2
2	4
3	6
4	8
5	10
6	12
7	14
8	16

The sum

Given `fR`, we want to take the sum of its range. We can do that by summing each of its range values - one by one - in the **order** of their associated domain values.

Divide and conquer

Another, more fruitful approach, is to recursively split the sum 'problem' into two (equally sized) sub-problems. This is called **divide and conquer**.

To make that work, we need to be able to quickly split a binary relation into two parts of roughly equal size. We also want to know the size of each relation, so that we can stop recursion when we have reached a single pair. At that point, we pick the range value of that single pair.

Here is some pseudo scala code that implements the divide and conquer summation of `fR`:

```
sum(fR: Relation[Int,Int]): Int = {
 if (fR.size == 1) fR.firstInRange
 else {
  val (left,right) = fR.split
  sum(left) + sum(right)
 }
}
```

Memoizing sum

To incrementally re-use sums we also memoize the sum function, represented by relation `sR`.
After calling `sum(fR)` we end up with the following memoization relation `sR`:

sR*	
2	2
4	4
6	6
8	8
10	10
12	12
14	14
16	16

sR*	
2,4	6
6,8	14
10,12	22
14,16	30

sR*	
2,4,6,8	20
10,12,14,16	52

sR*	
2,4,6,8,10,12,14,16	72

*Note that the 4 relations in the above example must be interpreted as a single version.

New versions

To demonstrate the efficient re-use of computations, we create a new version - named `f1` - which is based on `f`. This new version 'replaces' `8->16` with *`8->100`*.

f1R	
1	2
2	4
3	6
4	8
5	10
6	12
7	14
8	***100***

Consequently, we also create a new version `f1R` which is based on `fR`. After calling `sum(f1R)` the memoization relation `sR` is extended to `s1R` with 4 additional pairs.

s1R*	
2	2
4	4
6	6
8	8
10	10
12	12
14	14
16	16
100	***100***

s1R*	
2,4	6
6,8	14
10,12	22
14,16	30
14,100	***114***

s1R*	
2,4,6,8	20
10,12,14,16	52
10,12,14,100	***136***

s1R*	
2,4,6,8,10,12,14,16	72
2,4,6,8,10,12,14,100	***156***

Nothing is destroyed while creating new versions. All versions co-exist and structurally share the same data. Note that only a small amount of `O(log n)` pairs are added when the memoization relation `sR` is extended to `s1R`.

Creating versions

To achieve maximum sharing of data, and their derivations such as sums, we implement binary relations via **uniquely represented confluently persistent treaps** [1].

With confluently persistent treaps, replacing a single pair in `fR` only takes `O(log n)` to produce `f1R`.
In similar vein, it takes $O(\log^2 n)$ to go from `sR` to `s1R`.

Splitting a tree

When a relation is represented by a **balanced** binary tree the split operation can be done in `O(1)` time. The split operation just takes the left- and right node of the parent node that is split.

Because a treap is also a balanced binary tree, its left and right nodes must have roughly the same size, i.e. they roughly contain the same number of leafs.
By design, it is exactly this balancing property of treaps that is used to steer the divide and conquer process.

Uniqueness

Because confluently persistent treaps can also be uniquely represented, they are the basis for maximum re-use of data - and computations via memoization.

Discussion

Divide and conquer techniques can be applied to many algorithms, especially algorithms that involve aggregates over vector data.
Of course, we cannot always reformulate an algorithm in a divide and conquer style.

References

[1] R. van Dalen: Finite higher-ordered binary relations.
http://github.com/rapido/spread/blob/master/binary_relations_2012.pdf