Master's Thesis Proposal

Memoization of Incremental Computation for Generic Data Types

Jort van Gorkum

February 8, 2022

Student: Jort van Gorkum
Student number: 6142834
Master: Computing Science
Track: Programming Technology

University supervisor: Dr. Wouter Swierstra
2nd University supervisor: Dr. Trevor McDonell

Todo list

1 Introduction

- What is the problem? Illustrate with an example.

Example Haskell Data Type

Calculate a result over the Data Type

Cache the incremental computation over the data type

1.1 Research Questions

- What is/are your research questions/contributions?

What type of data structures / dsl can be used to keep track of the incremental computations

2 Background

2.1 An Efficient Algorithm for Type-Safe Structural Diffing

The paper *An Efficient Algorithm for Type-Safe Structural Diffing* by Victor Cacciari Miraldo and Wouter Swierstra presents an efficient datatype-generic algorithm called *hdiff* to compute the difference between two values of any algebraic datatype. In particular, the algorithm readily works over the abstract syntax tree (AST) of a programming language[8].

The algorithm when implemented in Haskell contains two main functions the `diff` and `apply`. The `diff` function computes the difference between two values of type `a`, and the `apply` function attempts to transform one value according to the information stored in the `Patch`.

```
diff  :: a -> a -> Patch a
apply :: Patch a -> a -> Maybe a
```

These functions are expected to fulfill some properties. The first being *correctness*: the patch that `diff x y` computes can be used to faithfully reproduces `y` from `x`.

```
∀ x y . apply(diff x y) x ≡ Just y
```

The second being *preciseness*:

```
∀ x y . apply(diff x x) y ≡ Just y
```

The last being *computationally efficient*: both the `diff` and `apply` functions needs to be space and time efficient.

The most commonly used diffing algorithm by version control systems is the Hunt-McIlroy algorithm used by the UNIX `diff` utility[5]. The UNIX `diff` satisfies these previously stated properties for `a ≡ [String]`[8]. Several attempts have been made to generalize this algorithm for arbitrary datatypes, but the way the UNIX `diff` represents the `Patch` using only *insertions, deletions* and *copies of lines* has two weaknesses. Firstly, the non-deterministic nature of the design makes the algorithm inefficient, and secondly, there exists no canonical 'best' patch and the choice is arbitrary[8].

Miraldo's and Swierstra's algorithm improves this shortcoming by introducing more operations: *arbitrary reordering, duplication* and *contraction of subtrees*. This restricts non-determinism, making it easier to compute patches and increasing the opportunities for copying.

To make the `diff` algorithm work, an implementation of *which common subtree* needs to be defined. The `wcs` function is a function that when given two trees and a subtree, returns the position of the subtree inside the trees if both contain the subtree. Otherwise, the function returns nothing. An example of a naive implementation would be:

```
wcs :: Tree -> Tree -> Tree -> Maybe Int
wcs s d x = elemIndex x (subtrees s ∩ subtrees d)
```

The paper identifies two inefficiencies using this naive implementation. (A) Checking trees for equality is linear in the size of the tree; (B) Furthermore, enumerating all subtrees is exponential.

To improve the first inefficiency of the naive `wcs` implementation is to use cryptographic hash functions to compare the equality of the trees. To check the trees for equality in constant time the trees are decorated with a hash at every node in the tree. Then, using the precomputed hash and the root node of the given tree, the hash of a subtree is calculated in constant time.

The second inefficiency of the naive `wcs` implementation is improved by using a `Trie`[1] datastructure.

2.2 Sums of Products for Mutually Recursive Datatypes

The paper *Sums of Products for Mutually Recursive Datatypes* written by Victor Cacciari Miraldo and Alejandro Serrano[7] presents a new approach to generic programming using recursive positions to handle mutually recursive families and the *sum-of-products* structure. This work (`generics-msrop`) is later used by the paper *An Efficient Algorithm for Type-Safe Structural Diffing* by Victor Cacciari Miraldo and Wouter Swierstra[8] to define the generic version of their diffing algorithm. Compared to existing generic programming libraries, `generics-mrsop` has *deep explicit recursion, sums of products* and supports *mutually recursive datatypes.*

Explicit recursion There are two ways to represent values. One contains the information on what properties of a datatype are recursive. The other does not contain that information. If we do not know explicitly if the property is recursive, then only one layer of the value can be formed into a generic representation. This is called *shallow* encoding. If we explicitly keep track of the recursive property, then the entire value can be transformed into a generic representation. This is called *deep* encoding. Using the *deep* encoding more datatypes can be defined generically (e.g., a generic *map* or generic Zipper datatype).

Sums of Products The `generic-sop` library uses a list of lists of types. The outer list represents the sum and the inner list represents the product. The sum represents the choice between two constructors; the product represents a combination of two constructors. An example of a `Code` representation of a `BinTree` is

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

Code_BinTree(Bin a) = `[`[a], `[Bin a, Bin a]]
```

Here the ` sign in the code promotes the definition to the type-level instead of a run-time value. The use of *Sums of Products* makes it considerably easier to represent generic datatypes.

Mutually recursive datatypes Most of the generic programming libraries are restricted to only allowing recursion on the same datatype, which is the one being defined. Mutually recursive datatypes are recursively defined in each other's terms. This means that most generic programming libraries do not support mutually recursive datatypes. Which limits the ability to generically represent the syntax of many programming languages. Thus `generic-sop` introduces recursive positions on a type level, which can be used to define mutually recursive datatypes.

2.3 Concise, Type-Safe, and Efficient Structural Diffing

The paper *Concise, Type-Safe, and Efficient Structural Diffing* written by Erdweg, Sebastian and Szabó, Tamás and Pacak, André presents a structural diffing algorithm called *truediff* [3]. *truediff* ensures that the patches produces are concise and type safe, and with a performance by an order of magnitude higher than Gumtree[4] and the *hdiff* [8] algorithm.

To compute the difference between a source tree and a target tree, *truediff* operates in four steps: (1) prepare subtree equivalence relations; (2) find reusable candidates; (3) select reusable candidates; (4) and compute the edit script.

The equivalence relations used in step 1, exist out of two equivalence relations, both encoded through cryptographic hashes. The first equivalence relation is used to identify reusable candidates. The second equivalence relation is used to identify preferred reusable candidates. The paper found that using structural equivalence to identify candidates and literal equivalence to select preferred candidates yields very concise edit scripts.

Describe how hdiff compares

3 Preliminary Results

Before writing the algorithm using the generic library `generic-msrop`[7], the algorithm is written using simpler self-defined generic datatypes with a fixpoint, which are defined in Appendix A and B. An example of how the generic datatypes can be used is:

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

type TreeG a = Fix (TreeF a)
type TreeF a = K a                    -- Leaf
            :+: ((I :*: K a) :*: I)   -- Node
```

Using the generic datatypes a `merkle` function can be defined, where at every recursive step of the datatype a `Hash` is stored. To merkelize a datatype, the datatype has to have the `Merkelize` constraint. The `Merkelize` type class is a class containing a single function `merkleIn` which converts the once unpacked `Fix` datatype into a unpacked `Fix` which contains a Hash at every recursive step[1].

```
merkle :: Merkelize f => Fix f -> Fix (f :*: K Hash)
merkle = In . merkleIn . unFix
```

```
class (Functor f) => Merkelize f where
  merkleIn :: (Merkelize g)
           => f (Fix g) -> (f :*: K Hash) (Fix (g :*: K Hash))
```

The generic datatypes can also use a `cata` function. The `cata` or catamorphism is a generalization of the concept of a fold, which means it deconstructs a data structure into its underlying functor[2].

```
cata :: Functor f => (f a -> a) -> Fix f -> a
cata alg t = alg (fmap (cata alg) (unFix t))
```

The `cata` function can then be used to, for example, calculate the sum of all the values of the nodes and the leaves of the tree.

```
cataSum :: TreeG Int -> Int
cataSum = cata (\case
  Inl (K x)                       -> x
  Inr (Pair (Pair (I l, K x), I r)) -> l + x + r)
```

[1]The implementation of the generic datatypes for the `Merkelize` type class can be found in Appendix C.

To keep track of the incremental computation of the summation of the tree, a HashMap[6] is used. The calculation of the incremental step is inserted into the HashMap and a pair of the HashMap and the result is returned. The implementation for the `TreeG` datatype is:

```haskell
cataMerkleTree :: TreeG Int -> (Map Hash Int, Int)
cataMerkleTree t = cata sumTree merkleTree
  where
    merkleTree :: Fix (TreeF a :*: K Hash)
    merkleTree = merkle t

    sumTree :: (TreeG Int :*: K Hash) Int -> Int
    sumTree (Pair (px, K h)) = case px of
      -- Leaf
      Inl (K x)
        -> (M.insert h x M.empty, x)
      -- Node
      Inr (Pair (Pair (I (xl, ml), K x), I (xr, mr)))
        -> let n = x + xl + xr
           in (M.insert h n (ml <> mr), n)
```

The problem with this implementation is that it only works for the `TreeG` datatype. A generic definition for the should look something like this:

Define the implementation where the intermediate results are automatically generated

```haskell
class CataMerkle f where
    cataMerkle :: (f a -> a) -> Fix (f :*: K Hash) -> (Map Hash a, a)
    cataMerkleWithMap :: (f a -> a) -> Map Hash a -> Fix (f :*: K Hash)
                      -> (Map Hash a, a)
```

Then using the previously generated HashMap, we can then calculate the result reusing the previously incremental computations:

```
cataMerkleTreeWithMap :: Map Hash Int -> TreeG Int -> (Int, Map Hash Int)
cataMerkleTreeWithMap m (In (Pair (x, K h))) =
  case lookup h m of
    Just n  -> (n, m)
    Nothing -> case x of
      Inl (K x) -> (x, insert h x empty)
      Inr (Pair (Pair (I l, K x), I r)) -> (x', m')
        where
          (xl, ml) = cataMerkleTreeWithMap m l
          (xr, mr) = cataMerkleTreeWithMap ml r
          x' = x + xl + xr
          m' = insert h x' mr
```

Using the previously defined cata functions we can determine the performance of the functions by using the `criterion`[9] package. For a benchmark in `criterion`, first the environment is setup. Then the bench function is executed multiples times within a certain timeframe. The result of the multiple executions is used to calculate the mean and standard deviation of the time executed.

The results of the `cataSum`, `cataMerkleTree` and `cataMerkleTreeWithMap` is seen in the graph.

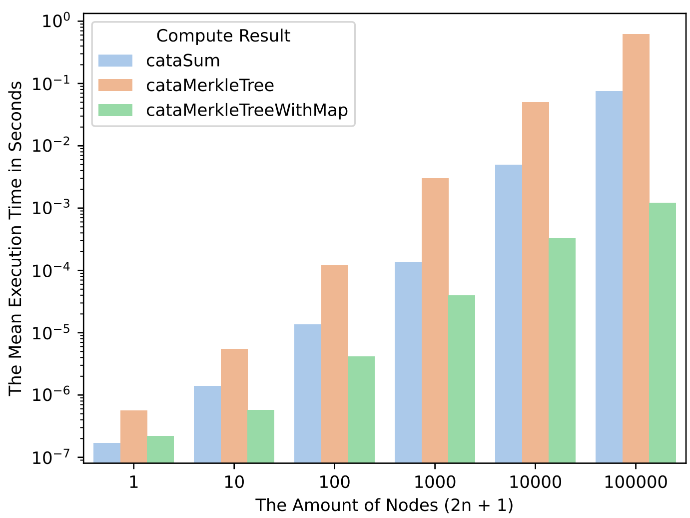

Figure 1: Compute the result

- What examples can you handle already?
- How can I generalize these results? What problems have I identified or do I expect?

4 Timetable and Planning

- What will I do with the remainder of my thesis?
- Give an approximate estimation/timetable for what you will do and when you will be done.

Writing a generic library to automatically cache the incremental computation

Which parameters can be tweaked to have the best performance / memory

What type of equivalence do you need to reuse incremental computation

What type of datastructures are the best to use for caching

5 Appendix

A Definition Generic Datatypes

```haskell
data I r          = I r
data K a r        = K a
data (:+:) f g r = Inl (f r) | Inr (g r)
data (:*:) f g r = Pair (f r, g r)
```

B Definition Fixpoint

```haskell
data Fix f = In { unFix :: f (Fix f) }

instance Eq (f (Fix f)) => Eq (Fix f) where
    f == g = unFix f == unFix g

instance Show (f (Fix f)) => Show (Fix f) where
    show = show . unFix
```

C Implementation Merkelize

```haskell
instance (Show a) => Merkelize (K a) where
  merkleIn (K x) = Pair (K x, K h)
    where
      h = hashConcat [hash "K", hash x]

instance Merkelize I where
  merkleIn (I x) = Pair (I prevX, K h)
    where
      prevX@(In (Pair (_, K ph))) = merkle x
      h = hashConcat [hash "I", ph]

instance (Merkelize f, Merkelize g) => Merkelize (f :+: g) where
  merkleIn (Inl x) = Pair (Inl prevX, K h)
    where
      (Pair (prevX, K ph)) = merkleIn x
      h = hashConcat [hash "Inl", ph]
  merkleIn (Inr x) = Pair (Inr prevX, K h)
```

```haskell
    where
      (Pair (prevX, K ph)) = merkleIn x
      h = hashConcat [hash "Inr", ph]

instance (Merkelize f, Merkelize g) => Merkelize (f :*: g) where
  merkleIn (Pair (x, y)) = Pair (Pair (prevX, prevY), K h)
    where
      (Pair (prevX, K phx)) = merkleIn x
      (Pair (prevY, K phy)) = merkleIn y
      h = hashConcat [hash "Pair", phx, phy]
```

D Results of computing the sum of a Tree

Amount	Action	Mean	Stddev
1	Generate (Result, Map)	5.662e-07	1.195e-08
1	Generate (Result, Map) with Map	2.208e-07	5.237e-09
1	Generate Result	1.713e-07	1.721e-09
10	Generate (Result, Map)	5.456e-06	6.462e-08
10	Generate (Result, Map) with Map	5.744e-07	8.788e-09
10	Generate Result	1.401e-06	1.132e-08
100	Generate (Result, Map)	1.205e-04	2.379e-06
100	Generate (Result, Map) with Map	4.165e-06	6.188e-08
100	Generate Result	1.358e-05	1.826e-07
1000	Generate (Result, Map)	3.024e-03	9.485e-05
1000	Generate (Result, Map) with Map	3.955e-05	6.024e-07
1000	Generate Result	1.387e-04	1.708e-06
10000	Generate (Result, Map)	5.018e-02	2.108e-03
10000	Generate (Result, Map) with Map	3.280e-04	5.800e-06
10000	Generate Result	4.994e-03	1.123e-04
100000	Generate (Result, Map)	6.253e-01	2.174e-02
100000	Generate (Result, Map) with Map	1.228e-03	1.836e-05
100000	Generate Result	7.592e-02	1.505e-03

Table 1: Compute the result

References

[1] Peter Brass. *Advanced data structures*. Vol. 193. Cambridge University Press Cambridge, 2008, pp. 336–356.

[2] *Catamorphisms*. URL: `https://wiki.haskell.org/Catamorphisms` (visited on Jan. 26, 2022).

[3] Sebastian Erdweg, Tamás Szabó, and André Pacak. "Concise, type-safe, and efficient structural diffing". In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 2021, pp. 406–419.

[4] Jean-Rémy Falleri et al. "Fine-grained and accurate source code differencing". In: *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. 2014, pp. 313–324.

[5] James Wayne Hunt and M Douglas MacIlroy. *An algorithm for differential file comparison*. Bell Laboratories Murray Hill, 1976.

[6] *Map type*. URL: `https://hackage.haskell.org/package/containers-0.4.0.0/docs/Data-Map.html` (visited on Feb. 8, 2022).

[7] Victor Cacciari Miraldo and Alejandro Serrano. "Sums of products for mutually recursive datatypes: the appropriationist's view on generic programming". In: *Proceedings of the 3rd ACM SIGPLAN International Workshop on Type-Driven Development*. 2018, pp. 65–77.

[8] Victor Cacciari Miraldo and Wouter Swierstra. "An efficient algorithm for type-safe structural diffing". In: *Proceedings of the ACM on Programming Languages* 3.ICFP (2019), pp. 1–29.

[9] Bryan O'Sullivan. URL: `http://www.serpentine.com/criterion/` (visited on Feb. 8, 2022).