Incremental Type-Safe Structural Diffing

Jort van Gorkum

January 19, 2022

Todo list

Write a piece about how the diff function uses an oracle which uses cryptographic hashes 2
Describe briefly the inner workings of the algorithm 4
Describe how hdiff compares 4

1 Introduction

- What is the problem? Illustrate with an example.
- What is/are your research questions/contributions?

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[4].

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[3]. The UNIX `diff` satisfies these previously stated properties for `a ≡ [String]`[4]. 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[4].

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.

Write a piece about how the diff function uses an oracle which uses cryptographic hashes

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[4] 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[4] 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* [1]. *truediff* ensures that the patches produces are concise and type safe, and with a performance by an order of magnitude higher than Gumtree[2] and the *hdiff* [4] algorithm.

Describe briefly the inner workings of the algorithm

Describe how hdiff compares

3 Preliminary Results

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

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

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

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


type TreeG a = Fix (TreeF a)
type TreeF a = K a
            :+: ((I :*: K a) :*: I)

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

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

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.

5 Appendix

A Definition Generic Datatypes

```
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

```
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

```
instance (Show a) => Merkelize (K a) where
  merkleIn (K x) = Pair (K x, K h)
    where
      h = digestConcat [digest "K", digest x]

instance Merkelize I where
  merkleIn (I x) = Pair (I prevX, K h)
    where
      prevX@(In (Pair (_, K ph))) = merkle x
      h = digestConcat [digest "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
```

```haskell
      h = digestConcat [digest "Inl", ph]
  merkleIn (Inr x) = Pair (Inr prevX, K h)
    where
      (Pair (prevX, K ph)) = merkleIn x
      h = digestConcat [digest "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 = digestConcat [digest "Pair", phx, phy]
```

D Implementation Cata Sum with Intermediate Results

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

    sumTree :: (TreeG Int :*: K Digest) 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)
```

References

[1] 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.

[2] 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.

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

[4] 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.