Summary - Related Work

Jort van Gorkum

December 9, 2021

Chapter 1

An Efficient Algorithm for Type-Safe Structural Diffing

1.1 Abstract

Effectively computing the difference between two versions of a source file has become an indispensable part of software development. The de facto standard tool used by most version control systems is the UNIX diff utility, that compares two files on a line-by-line basis without any regard for the structure of the data stored in these files. This paper presents an alternative datatype generic algorithm for computing the difference between two values of any algebraic datatype. This algorithm maximizes sharing between the source and target trees, while still running in linear time. Finally, this paper demonstrates that by instantiating this algorithm to the Lua abstract syntax tree and mining the commit history of repositories found on GitHub, the resulting patches can often be merged automatically, even when existing technology has failed.

1.2 Introduction

- A consequence of the by line granularity of the UNIX diff is it inability to identify more fine-grained changes in the objects it compares.
- Ideally, however, the objects under comparison should dictate the granularity of change to be considered. This is precisely the goal of structural differencing tools.
- In this paper we present an efficient datatype-generic algorithm to compute the difference between two elements of any mutually recursive family
- The *diff* function computes these differences between two values of type a,
- and *apply* attempts to transform one value according to the information stored in the *Patch* provided to it.
- We expect certain properties of our diff and apply functions.
  - The first being *correctness*: the patch that $diff\ x\ y$ computes can be used to faithfully reproduces y from x.

$$\forall\, x\ y\ .\ apply(diff\ x\ y)\ x\ \equiv\ Just\ y$$

  - The second being *preciseness*:

$$\forall\, x\; y\; .\; apply(diff\; x\; x)\; y \;\equiv\; Just\; y$$

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

- There have been several attempts at generalizing UNIX diff results to handle arbitrary datatypes, but following the same recipe: enumerate all combinations of insertions, deletions and copies that transform the source into the destination and choose the 'best' one. We argue that this design has two weaknesses when generalized to work over arbitrary types:
  - The non-deterministic nature of the design makes the algorithms inefficient.
  - There exists no canonical 'best' patch and the choice is arbitrary.
- This paper explores a novel direction for differencing algorithms: rather than restricting ourselves to *insertions, deletions,* and *copy operations*, we allow the *arbitrary reordering, duplication,* and *contraction of subtrees.*

1.3 Tree Diffing: A Concrete Example

- We explicitly model permutations, duplications and contractions of subtrees within our notion of *change*. Where contraction here denotes the partial inverse of a duplication.
- The representation of a *change* between two values of type `Tree23`, is given by identifying the bits and pieces that must be copied from source to destination making use of permutations and duplications where necessary.
- A new datatype `Tree23C` φ, enables us to annotate a value of `Tree23` with holes of φ. Therefore, `Tree23C MetaVar` represents the type of `Tree23` with holes carrying metavariables.
- These metavariables correspond to arbitrary trees that are *common subtrees* of both the source and destination of change.
- We refer to a value of `Tree23C` as a *context*.
- A *change* in this setting is a pair of such contexts. The first context defines a pattern that binds some metavariables, called the **deletion context**; the second, called the **insertion context**, corresponds to the tree annotated with the metavariables that are supposed to be instantiated by the binding given by the deletion context.

```
type Change23 φ = (Tree23C φ, Tree23C φ)
```

- Applying a change is done by instantiating the metavariables in the deletion context and the insertion context:

```
applyChange :: Change23 MetaVar -> Tree23 -> Maybe Tree23
applyChange (d, i) x = del x >>= ins i
```

- The changeTree23 function merely has to compute the deletion and insertion contexts

```
changeTree23 :: Tree23 -> Tree23 -> Change23 MetaVar
changeTree23 s d = (extract (wcs s d) s, extract (wcs s d) d)
```

- The extract function receives an oracle and a tree. It traverses its argument tree, looking for opportunities to copy subtrees.

```
extract :: (Tree23 -> Maybe MetaVar) -> Tree23 -> Tree23C MetaVar
extract o t = maybe (peel t) Hole (o t)
    where peel Leaf = LeafC
          peel (Node2 a b)
            = Node2C (extract o a) (extract o b)
          peel (Node3 a b c)
            = Node3C (extract o a) (extract o b) (extract o c)
```

- This iteration of the changeTree23 function has a subtle bug: not all common subtrees can be copied. In particular, we cannot copy a tree t that occurs as a subtree of the source and destination, but also appears as a subtree of another, larger common subtree.
- One way to solve this is to introduce an additional post-processing step that substitutes the variables that occur exclusively in the deletion or insertion context by their corresponding tree.

```
changeTree23 :: Tree23 -> Tree23 -> Change23 MetaVar
changeTree23 s d
    = postprocess s d (extract (wcs s d) s) (extract (wcs s d) d)
```

1.3.1 Minimizing Changes

- The process of minimizing and isolating the changes starts by identifying the redundant part of the contexts. That is, the constructors that show up as a prefix in both the deletion and the insertion context.
- They are essentially being copied over, and we want to make this fact explicit by separating them into what we call the *spine* of the patch.
- If a constructor is in the spine, we know it has been copied, if it shows up in a change, we know it was either deleted or inserted.

```
type Patch23 = Tree23C (Change23 MetaVar)

patch :: Patch23
patch = Node3C (Hole (Hole 0, Hole 0))
               (Hole (Node2C (Hole 0) (Hole 1)
                     , Node2C (Hole 1) (Hole 0)))
               (Node2C (Hole (tree23ToC w, tree23ToC w'))
                       (Hole (Hole 3, Hole 3)))
```

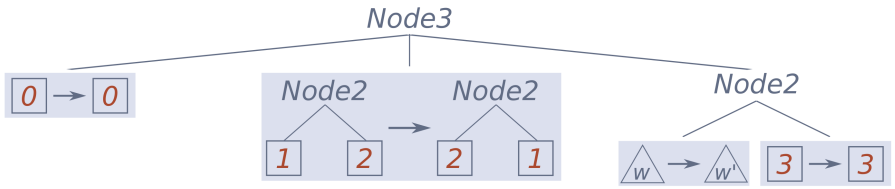

Figure 1.1: Patch23 Example

- A patch consists in a spine with changes inside it. Figure 1.1 illustrates a value of type Patch23, where the changes are visualized with a shaded background in the leaves of the spine.
- The first step to compute a patch from a change is identifying its spine.
- We are essentially splitting a monolithic change into the greatest common prefix of the insertion and deletion contexts, leaving smaller changes on the leaves of this prefix:

```
gcp :: Tree23C var -> Tree23C var -> Tree23C (Change23 var)
gcp LeafC LeafC = LeafC
gcp (Node2C a1 b1) (Node2C a2 b2)
    = Node2C (gcp a1 a2) (gcp b1 b2)
gcp (Node3C a1 b1 c1) (Node3C a2 b2 c2)
    = Node3C (gcp a1 a2) (gcp b1 b2) (gcp c1 c2)
```

- The greatest common prefix consumes all the possible constructors leading to disagreeing parts of the contexts where this might be too greedy.
- To address this problem, we go over the result from our call to `gcp`, pulling changes up the tree until each change is closed, that is, the set of variables in both contexts is identical. We call this process the closure of a patch
- The final diff function for `Tree23` is then defined as follows:

```
diffTree23 :: Tree23 -> Tree23 -> Patch23
diffTree23 s d = closure $ gcp $ changeTree23 s d
```

1.3.2 Defining the `wcs` for `Tree23`

- In order to have a working version of our diff algorithm for Tree23 we must provide the wcs implementation. Recall that the wcs function, *which common subtree*, has type:

```
wcs :: Tree23 -> Tree23 -> Tree23 -> Maybe MetaVar
```

- Given a fixed s and d, *wcs* s d x returns Just i if x is the i^{th} subtree of s and d and Nothing if x does not appear in s or d.
- To tackle the first issue and efficiently compare trees for equality we will be using cryptographic hash functions to construct a fixed length bitstring that uniquely identifies a tree modulo hash collisions.

- Said identifier will be the hash of the root of the tree, which will depend on the hash of every subtree, much like a Merkle tree

```
merkleRoot :: Tree23 -> Digest
merkleRoot Leaf = emptyDigest
merkleRoot (Node2 x y)
    = hash (concat ["node2", merkleRoot x, merkleRoot y])
merkleRoot (Node3 x y z)
    = hash (concat ["node3", merkleRoot x, merkleRoot y, merkleRoot z])
```

- the ($\equiv$) definition above is still linear, we recompute the hash on every comparison. We fix this by caching the hash associated with every node of a Tree23. This is done by the decorate function

```
data Tree23H = LeafH
             | Node2H (Tree23H, Digest) (Tree23H, Digest)
             | Node3H (Tree23H, Digest)
                      (Tree23H, Digest)
                      (Tree23H, Digest)
```

- This enables us to define a constant time merkleRoot function, shown below, which makes the ($\equiv$) function run in constant time.

```
merkleRoot :: Tree23H -> Digest
merkleRoot LeafH = emptyDigest
merkleRoot (Node2H (_, hx) (_, hy))
    = hash (encode "2" ++ hx ++ hy)
merkleRoot (Node3H (_, hx) (_, hy) (_, hz))
    = hash (encode "3" ++ hx ++ hy ++ hz)
```