An Efficient Algorithm for Type-Directed Structural Diffing

VICTOR CACCIARI MIRALDO, Utrecht University, The Netherlands
WOUTER SWIERSTRA, Utrecht University, The Netherlands

Effectively computing the difference between two version 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.

Additional Key Words and Phrases: Generic Programming, diff, Version Control, Haskell

1 INTRODUCTION

The UNIX `diff` [13] is an essential tool in modern software development. It has seen a number of use cases ever since it was created and lies at the heart of today's Software Version Control Systems. Tools such as git, mercurial and darcs, that enable multiple developers to collaborate effectively, are all built around the UNIX `diff` utility, which is used to compute a patch between two versions of a file. It compares files on a line-by-line basis attempting to share as many lines as possible between the source and the destination files.

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. For example, if two parts of a program were changed, but happen to be printed on the same line, the UNIX `diff` sees this as a *single* change. 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. In particular, our algorithm readily works over the abstract syntax tree of a programming language— thereby enabling, for example, two changes that work on separate parts of the AST to be trivially merged, even if they appear to be on the same line in the source file. We have implemented our algorithm in Haskell and make heavy use of its datatype generic programming capabilities.

In general, we intend to compute the difference between two values of type a, and represent these changes in some type, $Patch\ a$. 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.

$$\begin{aligned} diff &:: a \to a \to Patch\ a \\ apply &:: Patch\ a \to a \to Maybe\ a \end{aligned}$$

Note that the *apply* function may fail, for example, when attempting to delete data that is not present. Yet when it succeeds, the *apply* function must return a value of type a. This may seem like an obvious design choice, but this property does not hold for the approaches [3, 10] using `xml` or

Authors' addresses: Victor Cacciari Miraldo, Information and Computing Sciences, Utrecht University, Princetonplein, 5, Utrecht, Utrecht, 3584 CC, The Netherlands, v.cacciarimiraldo@uu.nl; Wouter Swierstra, Information and Computing Sciences, Utrecht University, Princetonplein, 5, Utrecht, Utrecht, 3584 CC, The Netherlands, w.s.swierstra@uu.nl.

2017. XXXX-XXXX/2017/3-ART $15.00
https://doi.org/

`json` to represent their abstract syntax trees, where the result of applying a patch may produce ill-typed results.

Naturally, not every definition of *Patch*,*diff* and *apply* will solve our problem. 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 *apply* function is inherently partial and *correctness* only requires apply to succeed in one particular instance—but what should happen when applying a patch to a different value than the one used to create the input patch? We argue that the apply function should only fail when strictly necessary. In particular, if there are no changes, the patch should represent a *no-op*, and its application should be the identity:

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

This captures the idea that a patch that does not make any modifications must be applicable to any value.

Finally, the last important properties stem from a practical perspective. We need both the *diff* and *apply* functions to be computationally efficient. Lastly, when stored in disk, a value of type $Patch\ a$ must use less space then storing both elements of type a. Otherwise, one could argue that $Patch\ a = (a, a)$ is a perfectly fine solution. Yet, storing all revisions of every file under version control is clearly not an acceptable solution.

The UNIX `diff` [13] satisfies these properties for the specific type of lines of text, or, $a \equiv [String]$. It represents patches as a series of insertions, deletions and copies of lines and works by enumerating all possible patches that transform the source into the destination and chooses the 'best' such patch. There have been several attempts at a generalizing these results to handle arbitrary datatypes [17, 23], 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: (A) the non-deterministic nature of the design makes the algorithms inefficient, and (B), there exists no canonical 'best' patch and the choice is arbitrary.

We illustrate this last point with the example in Figure 1. The existing datatype generic approaches with insertions, deletions and copies typically perform a preorder traversal of the trees, copying over constructors whenever possible. Yet if we want to transform a binary tree $Bin\ t\ u$ into $Bin\ u\ t$ using only these operations, we will be forced to choose between copying t or u, but never both. The choice of which subtree to copy becomes arbitrary and unpredictable. To make matters worse, the non-determinism such choice points introduce makes algorithms intractably slow.

The central design decision underlying the UNIX `diff` tool is to *copy data whenever possible*. Yet this example shows that using only insertions, deletions and copies as the set of operations also limits the opportunities for copying data. In the presence of richly structured data beyond lines of text, this becomes especially problematic.

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. Not only does this restrict the inherent non-determinism in the problem, making it *easier* to compute patches, this also *increases* the opportunities for copying. More specifically, this paper makes the following novel contributions:

- This paper defines a datatype generic *diff* function that computes a patch between two algebraic datatypes that is efficient in both time and space. This *diff* function supports the

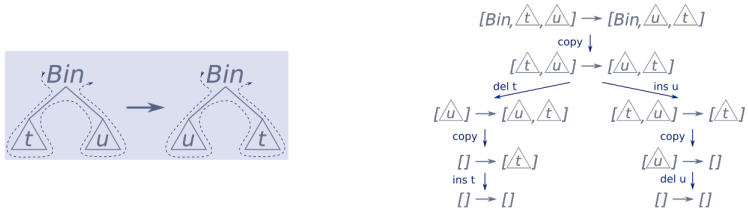


Fig. 1. Visualization of a *diff* (*Bin t u*) (*Bin u t*) using insertions, deletions and copies only

duplication and permutation of subtrees, and satisfies all the desired properties outlined above. We illustrate this algorithm by first defining a specific instance (Section 2), then presenting a generic version capable of handling arbitrary mutually recursive families of datatypes (Section 3).

- Initially, we present our diff algorithm assuming the existence of an oracle capable of detecting all possible copying opportunities. We give a practical implementation of this oracle that is correct modulo cryptographic hash collisions and runs in constant time (Section 4).
- We show how the representation for patches used in this paper enables *disjoint* patches to be *merged* automatically (Section 5).
- Finally, we have instantiated our algorithm to the abstract syntax tree of Lua and collected historical data regarding merge conflicts from popular GitHub repositories. We show how our naive merging algorithm is already capable of resolving more than 10% of the merge conflicts encountered automatically, while still offering competitive performance (Section 6).

2 TREE DIFFING: A CONCRETE EXAMPLE

Before exploring the generic implementation of our algorithm, let us look at a simple, concrete instance first. This example sets the stage for the the generic implementation that follows (Section 3.2). Throughout this section we will explore the central ideas from our algorithm instantiated for the type of 2-3-trees:

```
data Tree23 = Leaf
            | Node2 Tree23 Tree23
            | Node3 Tree23 Tree23 Tree23
```

The central concept of our work is the encoding of a *change*. Unlike previous work [15, 17, 23] which is based on tree-edit-distance [5] and hence, uses only insertions, deletions and copies of the constructors encountered during the preorder traversal of a tree (Figure 1), we go a step further. 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*, then, 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 type 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 the change. These are exactly the bits that are being copied from the source to the destination tree. We refer to a value of *Tree23C* as a *context*. For now, the metavariables will be simple *Int* values but later on we will need to carry additional information.

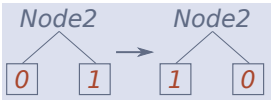


Fig. 2. Visualization of a *diff* (*Node2 t u*) (*Node2 u t*), metavariables are shown in red

```
type MetaVar = Int

data Tree23C φ = Hole φ
               | LeafC
               | Node2C Tree23C Tree23C
               | Node3C Tree23C Tree23C Tree23C
```

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 bindings given by the deletion context.

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

The change that transforms *Node2 t u* into *Node2 u t* is then represented by a pair of *Tree23C*, (*Node2C* (*Hole* 0) (*Hole* 1), *Node2C* (*Hole* 1) (*Hole* 0)), as seen in Figure 2. This change works on any tree built using the *Node2* constructor and swaps the children of the root. Note that it is impossible to define such swap operations using insertions and deletions—as used by most diff algorithms.

2.1 Applying Changes

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 d x >>= ins i
```

Naturally, if the term x and the deletion context d are *incompatible*, this operation will fail. Contrary to regular pattern-matching we allow variables to appear more than once on both the deletion and insertion contexts. Their semantics are dual: duplicate variables in the deletion context must match equal trees, and are referred to as contractions, whereas duplicate variables in the insertion context will duplicate trees. We use an auxiliary function within the definition of *del* to make this check easier to perform. Given a deletion context *ctx* and source *tree*, the *del* function tries to associate all the metavariables in the context with a subtree of the input *tree*.

```
del :: Tree23C MetaVar → Tree23 → Maybe (Map MetaVar Tree23)
del ctx tree = go ctx tree empty
  where
    go :: Tree23C → Tree23 → Map MetaVar Tree23 → Maybe (Map MetaVar Tree23)
    go LeafC          Leaf          m = return m
    go (Node2C x y)   (Node2 a b)   m = go x a m >=> go y b
    go (Node3C x y z) (Node3 a b c) m = go x a m >=> go y b >=> go z c
    go (Hole i)       t             m = case lookup i m of
                                        Nothing → return (M.insert i t m)
                                        Just t' → guard (t ≡ t') >> return m
    go _              _             m = Nothing
```

The *go* function above closely follows the structure of trees and contexts. Only when we reach a *Hole*, do we check whether we have already instantiated the metavariable stored there or not. If we have encountered this metavariable before, we check that both occurrences of the metavariable correspond to the same tree; if this is the first time we have encountered this metavariable, we simply instantiate the metavariable with the current tree. We will refer to the result of *del ctx tree* as the *valuation* that instantiates the metavariables of *ctx* with subtrees of *tree*.

Once we have obtained a such valuation, we substitute the variables in the insertion context with their respective values, to obtain the final tree. This phase fails when the change contains unbound variables.

```
ins :: Tree23C MetaVar → Map MetaVar Tree23 → Maybe Tree23
ins LeafC          m = return Leaf
ins (Node2C x y)   m = Node2 <$> ins x m <*> ins y m
ins (Node3C x y z) m = Node3 <$> ins x m <*> ins y m <*> ins z m
ins (Hole i)       m = lookup i m
```

2.2 Computing Changes

Next, we explore how to produce a change from a source and a destination, defining a *changeTree23* function. This function will exploit as many copy opportunities as possible. For now, we delegate the decision of whether a subtree should be copied or not to an oracle: assume we have access a function $ics : Tree23 \to Tree23 \to Tree23 \to Maybe\ MetaVar$, short for *"is common subtree"*. The call *ics s d x* returns *Nothing* when *x* is *not* a subtree of *s* and *d*; if *x* is a subtree of both *s* and *d*, it returns *Just i*, for some metavariable *i*. The only condition we impose is injectivity of *ics s d*: that is, if *ics s d x* ≡ *ics s d y* ≡ *Just j*, then $x \equiv y$. In other words, equal metavariables correspond to equal subtrees.

There is an obvious inefficient implementation for *ics*, that traverses both trees searching for shared subtrees—hence assuming the existence of such an oracle is not a particularly strong assumption to make. In Section 4, we provide an efficient and generic implementation. For now, assuming the oracle exists allows for a clear separation of concerns. The *changeTree23* function merely has to compute the deletion and insertion contexts, using said oracle—the inner workings of the oracle are abstracted away cleanly.

```
changeTree23 :: Tree23 → Tree23 → Change23 MetaVar
changeTree23 s d = (extract (ics s d) s, extract (ics s d) d)
```

The *extract* function receives an oracle and a tree. It traverses its argument tree, looking for opportunities to copy subtrees. It repeatedly consults the oracle, to determine whether or not the

$$a = Node2\ (Node2\ t\ k)\ u \qquad extract\ (ics\ a\ b)\ a = Node2C\ (Hole\ 0)\ u$$
$$b = Node2\ (Node2\ t\ k)\ t \qquad extract\ (ics\ a\ b)\ b = Node2C\ (Hole\ 0)\ (Hole\ 1)$$

$$postprocess\ a\ b\ (extract\ (ics\ a\ b)\ a)\ (extract\ (ics\ a\ b)\ b) = (Node2C\ (Hole\ 0)\ u, Node2C\ (Hole\ 0)\ t)$$

Fig. 3. Example of erroneous context extraction due to nested common subtrees

current subtree should be shared across the source and destination. If that is the case, we want our change to *copy* such subtree. That is, we return a *Hole* whenever the second argument of *extract* is a common subtree according to the oracle. If the oracle returns *Nothing*, we move the topmost constructor to the context being computed and recurse over the remaining 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)
```

Note that had we used a version of *ics* that only returns a boolean value we would not know what metavariable to use when a subtree is shared. Returning a value that uniquely identifies a subtree allows us to keep the *extract* function linear in the number of constructors in x (disregarding the calls to our oracle for the moment).

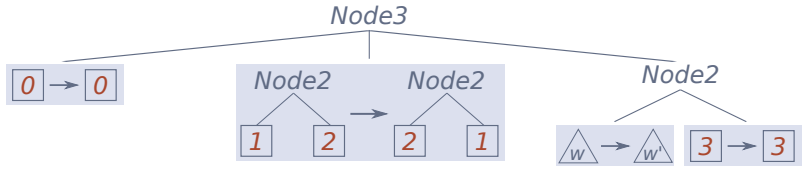
This iteration of the *changeTree23* function has a subtle bug: not all common subtrees can be copied. In particular, we cannot copy a tree t that occurs as a subtree of the source and destination, but also appears as a subtree of another, larger common subtree. One such example is shown in Figure 3, where the oracle claims that both $Node2\ t\ k$ and t are common subtrees. As t also occurs by itself one of the extracted contexts will contain an unbound metavariable. This will trigger an error when trying to apply the corresponding change. In this example, applying the change from Figure 3 would trigger such error when the *ins* function branch for the *Hole* constructor and attempts to lookup the tree associated with metavariable 1.

One way to solve this is to introduce an additional postprocessing step that substitutes the variables that occur exclusively in the deletion or insertion context by their corresponding tree. We can implement this postprocessing step using two calls to the *del* function we saw previously: one for the deletion context against the source tree and another for the insertion context against the destination tree. The resulting information is then used to replace *unused* or *undeclared* metavariables with the tree to which they correspond. We postpone the implementation until its generic incarnation in Section 3.2.

```
postprocess :: Tree23 → Tree23 → Tree23C MetaVar → Tree23C MetaVar
            → (Tree23C MetaVar, Tree23C MetaVar)
```

We fix the previous *changeTree23* by postprocessing the extracted contexts. The new version of *changeTree23* will only produce closed changes, where each deletion and insertion context have *the same set of metavariables*. Intuitively, this means that every variable that is declared is used and vice-versa.

```
changeTree23 :: Tree23 → Tree23 → Change23 MetaVar
changeTree23 s d = postprocess s d (extract (ics s d) s) (extract (ics s d) d)
```




```
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)))
```

Fig. 4. Graphical and textual representation of the patch that transforms the value *Node3 t (Node2 u v) (Node2 w x)* into the value *Node3 t (Node2 v u) (Node2 w′ x)*. The *tree23toC* function converts a *Tree23* into a *Tree23C* in the canonical way.

Assuming that *ics s d* correctly assigns metavariables to *all* common subtrees of *s* and *d*, it is not hard to see that our implementation already satisfies the specification we formulated in the introduction:

Correctness Assuming *ics* is correct,

$$\forall\ x\ y\ .\ applyTree23\ (diffTree23\ x\ y)\ x \equiv Just\ y$$

Preciseness Assuming *ics* is correct,

$$\forall\ x\ y\ .\ applyTree23\ (diffTree23\ x\ x)\ y \equiv Just\ y$$

Time Efficiency On the worst case, we perform one query to the oracle per constructor in our trees. Assuming *ics* to be a constant time function, our algorithm is linear on the number of constructors in the source and destination trees. We will define a version of *ics* in Section 4 that requires constant time.

Space Efficiency The size of a *Change23 MetaVar* is, on average, smaller than storing its source and destination tree completely. On the worst case, where there is no common subtree, they have the same size. This is also true of the Unix `diff` utility, when comparing two files that do not share a single line of text.

Although correct with respect to our specification, there is still room for improvement. A call to *changeTree23 x y* yields a *single Change23*, consisting of a pair of insertion and deletion contexts. When *x* and *y* resemble one another these contexts may store a great deal of redundant information as many constructors appearing in both contexts will be 'deleted', and then 'inserted'. To address this, we want to share information between the deletion and insertion contexts, where possible. Moreover, it is much easier to handle small and isolated changes when considering merging changes, as we will see in Section 5.

2.3 Minimizing Changes: Computing Patches

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. The spine will then contain changes—pairs of an insertion and deletion context—in its leaves:

```
type Patch23 = Tree23C (Change23 MetaVar)
```

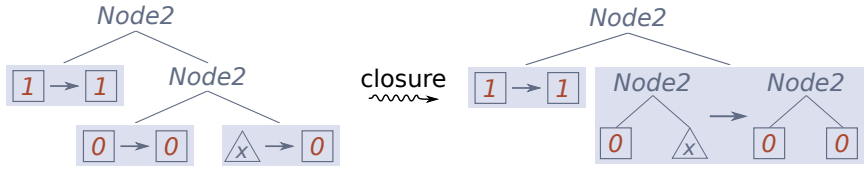


Fig. 5. Graphical representation of the *closure* function.

Figure 4 illustrates a value of type *Patch23*, where the *changes* are visualized with a shaded background in the leaves of the spine. Note that the changes encompass only the minimum number of constructor necessary to *bind and use* all metavariables. This keeps changes small and isolated.

On this section we will discuss how to take the results of *changeTree23* and transform them into a *Patch23*. The first step to compute a patch from a change is identifying its *spine*. That is, the constructors that are present in both the deletion and insertion contexts. 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:

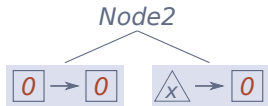
```
gcp :: Tree23C var → Tree23C var → Tree23C (Change23 var)
gcp LeafC                 LeafC                 = LeafC
gcp (Node2C a1 b1)        (Node2C a2 b2)        = Node2C (gcp a1 a2) (gcp b1 b2)
gcp (Node3C a1 b1 c1) (Node3C a1 b2 c2) = Node3C (gcp a1 a2) (gcp b1 b2) (gcp c1 c2)
gcp a                     b                     = Hole (a, b)
```

In the last case of the *gcp* function either a and b are both holes or the constructors disagree, hence they do *not* belong in the common prefix.

One might be tempted to take the results of *changeTree23C*, pipe them into the *gcp* function directly. Yet, the *greatest common prefix* consumes all the possible constructors leading to disagreeing parts of the contexts where this might be too greedy. We must be careful not to break bindings as shown below:

```
-- prob = changeTree23 (Node2 t t) (Node2 x t)
prob :: Change23 MetaVar
prob = Change (Node2C (Hole 0) (Hole 0)
              , Node2C x       (Hole 0))
```

gcp prob ≡



In this example, the second change contains a *Hole* 0 that does not occur in the deletion context, and is hence *unbound*. 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 and declare a function to compute that below.

We have illustrated the process of *closure* in Figure 5. Note that in both the input and output for the *closure* function the subtree x appears on the deletion context. Moreover, the *closure* functions only bubbles up the minimal number of constructors to ensure all changes are closed. In particular, the *Node2* constructor at the root is still part of the spine after the call to *closure*.

```
closure :: Tree23C (Change23 MetaVar) → Patch23
```

Although the *closure* function apparently always returns a patch, its implementation might fail if there exists no way of closing all the changes. In our case, this will never happen as we know that *changeTree23* outputs a closed change. In the worst case, the resulting spine will be empty—but

the change will certainly be closed. We will come back to the *closure* function in more detail on its generic incarnation (Section 3.2). For now, it is more important to understand the problem that it solves. As soon as every change within the spine has been closed, we have 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
```

We could define the *applyPatch23* function that applies a *patch*, rather than the *applyChange23* we saw previously. This is done by traversing the object tree and the spine of the patch until a change is found and applying that change to the *local* subtrees in question. Besides a having a localized application function, this representation with minimized changes enables us to trivially identify when two patches are working over disjoint parts of a tree. This will be particularly interesting when trying to *merge* different patches, as we will see shortly (Section 5).

3 TREE DIFFING GENERICALLY

In Section 2 we provided a simple algorithm to solve the differencing problem for 2-3-trees. We began by creating the type of contexts over *Tree23*, which consisted in annotating a *Tree23* with a *metavariable* constructor. Later, assuming the existence of an oracle that determines whether or not an arbitrary tree is a subtree of the source and the destination, we compute a value of type *Change23 MetaVar* from a *Tree23*. how to compute a *Patch23* given a *Change23* by *minimizing* the changes and isolating them in the *spine*. On this section we show how can we write that same algorithm in a generic fashion, working over any mutually recursive family. The code in this section generalizes the example from the previous section, but we assume some familiarity with generic programming in modern Haskell. Readers unfamiliar with these concepts may safely skip this section on a first reading.

3.1 Background on Generic Programming

Firstly, let us briefly review the `generics-mrsop` [24] library, that we will use to define a generic version of our algorithm. This library follows the *sums-of-products* school of generic programming [7] and, additionally, enables us to work with mutually recursive families. This is particularly important for this algorithm, as the abstract syntax trees of many programming languages consist of mutually recursive types for expressions, statements, methods, classes and other language constructs.

Take the *Tree23* type from Section 2. Its structure can be seen in a *sum-of-products* fashion through the *Tree23SOP* type given below. It represents the shape in which every *Tree23* comes and consists in two nested lists of *atoms*. The outer list represents the choice of constructor, and packages the *sum* part of the datatype whereas the inner list represents the *product* of the fields of a given constructor. The $'\cdot$ notation represents a value that has been promoted to the type level [31].

```
type Tree23SOP = '['[]
               , '[I 0, I 0]
               , '[I 0, I 0, I 0]]
```

The atoms, in this case only $I\ 0$, represent a recursive position referencing the first type in the family. In fact, a mutually recursive family is described by *a list of sums-of-products*: one for each element in the family. We overload the word "code" in singular or plural to mean the representation of a datatype, or the representation of a family, respectively. The context should make clear the distinction. For example, *Tree23SOP* is the code of type *Tree23* and *Tree23Codes* is the codes for the mutually recursive family.

```
data NS :: (k → ∗) → [k] → ∗ where
  Here  :: f x       → NS f (x ': xs)
  There :: NS f xs → NS f (x ': xs)

data NP :: (k → ∗) → [k] → ∗ where
  Nil  :: NP f '[]
  Cons :: f x → NP f xs → NP f (x ': xs)

data NA :: (Nat → ∗) → Atom → ∗ where
  NA_I :: φ i      → NA φ (I i)
  NA_K :: Opq k → NA φ (K k)
```

```
elimNS :: (∀ at . f at → a) → NS f s → a
elimNS f (There s) = elimNS f s
elimNS f (Here x)  = f x

elimNP :: (∀ at . f at → a) → NP f p → [a]
elimNP f Nil         = []
elimNP f (Cons x xs) = f x : elimNP f xs

elimNA :: (∀ ix . f x → a) → (∀ k . g k → a)
       → NA f g at → a
elimNA f g (NA_I x) = f x
elimNA f g (NA_K x) = g x
```

Fig. 6. Interpretation and elimination principles for each component of a sum-of-products code.

```
type Tree23Codes = '[Tree23SOP]
```

The `generics-mrsop` uses the type *Atom* to distinguish whether a field is a recursive position referencing the n-th type in the family, $I\ n$, or a opaque type, for example, *Int* or *Bool*, which are represented by $K\ KInt$, $K\ KBool$.

Let us now take a mutually recursive family with more than one element and see how it is represented within the `generics-mrsop` framework. The mutually recursive family containing types *Zig* and *Zag* has its codes defined as a list of codes, one for each member of the family:

```
data Zig = Zig Int  | ZigZag Zag
data Zag = Zag Bool | ZagZig Zig

type ZigCodes = '['['K KInt] , '[I 1]]
                ,'['K KBool], '[I 0]]
                ]
```

Note that the codes come in the same order as the elements of the family. The code for *Zig* is the first element of the *ZigCodes* type level list. It consists in two lists, since *Zig* has two constructors. One receives a value of type *Int*, the other consists in a recursive call to the second element of the family. The code acts as a recipe that the *representation* functor must follow in order to interpret those into a type of kind $*$.

The representation is defined by the means of n-ary sums (*NS*) and products (*NP*) that work by induction on the *codes* and one interpretation for atoms (*NA*). Their definition together with their respective elimination principles can be found in Figure 6.

The *NS* type is responsible for determining the choice of constructor whereas the *NP* applies a representation functor to all the fields of the selected constructor. Finally, *NA* distinguishes between representation of a recursive position from an opaque type. Although the `generics-mrsop` provides a way to customize the set of opaque types used, this is not central do the developments in this paper and hence we will assume a type *Opq* that interprets the necessary atom types, i.e., *Int*, *Bool*, etc. We refer the interested reader to the original paper [24] for more information. Nevertheless, we define the representation functor *Rep* as the composition of the interpretations of the different pieces:

```
type Rep φ = NS (NP (NA φ))
```

Finally, we tie the recursive knot with a functor of kind $Nat \to *$ that is passed as a parameter to NA in order to interpret the recursive positions. The familiar reader might recognize it as the indexed least fixpoint:

```
newtype Fix (codes :: '['['[Atom]]]) (ix :: Nat)
  = Fix {unFix :: Rep (Fix codes) (Lkup codes ix)}
```

Here, $Lkup\ codes\ ix$ denotes the type level lookup of the element with index ix within the list $codes$. This type family throws a type error if the index is out of bounds. The generic versions of the constructors of type Zig are given by:

```
gzig :: Int → Fix ZigCodes 0
gzig n = Fix (Here (Cons (NA_K (OpqInt n)) Nil))

gzigzag :: Fix ZigCodes 1 → Fix ZigCodes 0
gzigzag zag = Fix (There (Here (Cons (NA_I zag) Nil)))
```

One of the main benefits of the *sums-of-products* approach to generic programming is that it enables us to pattern match generically. In fact, we can state that a value of a type consists precisely of a choice of constructor and a product of its fields by defining a *view* type. For example, we take the *constructor* of a generic type to be:

```
data Constr :: [[k]] → Nat → * where
  CZ ::                Constr (x ': xs) Z
  CS :: Constr xs c → Constr (x ': xs) (S c)
```

The $Constr\ sum\ c$ type represents a predicate indicating that c is a valid constructor for sum, that is, it is a valid index into the type level list sum. This enables us to define a $View$ over a value of a sum type to be a choice of constructor and corresponding product. We can then unwrap a $Fix\ codes\ i$ value into its topmost constructor and a product of its fields with the sop function.

```
data View :: (Nat → *) → '['[Atom]] → * where
  Tag :: Constr sum c → NP (NA φ) (Lkup c sum) → View φ sum

sop :: Fix codes i → View (Fix codes) (Lkup i codes)
```

This brief introduction covers the basics of generic programming in Haskell that we will use in this paper. We refer the interested reader to the literature [7, 24] for a more thorough overview.

3.2 Representing and Computing Changes, Generically

Section 2 presented one particular instance of our differencing algorithm. In this section, we will generalize the definition using the generic programming techniques we have just seen.

We start defining the generic notion of context, called Tx. Analogously to $Tree23C$ (Section 2), Tx enables us to augment mutually recursive family with type holes. This type construction is crucial to the representation of patches. This construction can be done for any mutually recursive family.

We can read $Tx\ codes\ \varphi\ at$ as the element of the mutually recursive family $codes$ indexed by at augmented with holes of type φ. Its definition follows:

```
data Tx :: [[[Atom]]] → (Atom → ∗) → Atom → ∗ where
  TxHole :: φ at    → Tx codes φ at
  TxOpq  :: Opq k → Tx codes φ (K k)
  TxPeel :: Constr (Lkup i codes) c
         → NP (Tx codes φ) (Lkup (Lkup i codes) c)
         → Tx codes φ (I i)
```

Looking at the definition of Tx, we see that its values consist in either a *typed* hole, some opaque value, or a constructor and a product of fields. The $TxPeel$ follows very closely the $View$ type from Section 3.1. The Tx type is, in fact, the indexed free monad over the Rep. The return method is just $TxHole$, and the multiplication is given by:

```
txJoin :: Tx codes (Tx codes φ) at → Tx codes φ at
txJoin (TxHole tx)  = tx
txJoin (TxOpq opq) = TxOpq opq
txJoin (TxPeel c d) = TxPeel c (mapNP txJoin p)
```

Essentially, a value of type $Tx\ codes\ \varphi\ at$ is a value of type $NA\ (Fix\ codes)\ at$ augmented with *holes* of type φ. To illustrate this, let us define the injection of a NA into a Tx:

```
na2tx :: NA (Fix codes) at → Tx codes φ at
na2tx (NA_K k)       = TxOpq k
na2tx (NA_I (Fix x)) = case sop x of Tag c p → TxPeel c (mapNP na2tx p)
```

The inverse operations is partial. We can translate a Tx into an NA when the Tx has no *holes*:

```
tx2na :: Tx codes φ at → Maybe (NA (Fix codes) at)
tx2na (TxHole _)    = Nothing
tx2na (TxOpq k)     = NA_K k
tx2na (TxPeel c txs) = inj c <$> mapNPM tx2na txs
```

Generic Representation of Changes. With a generic notion of contexts, we can go ahead and define our generic $Change$ type. We use a pair of Tx exactly as in Section 2: one deletion context and one insertion context. This time, however, we define a new datatype to be able to partially apply its type arguments later on.

```
data Change codes φ at = Change (Tx codes φ at) (Tx codes φ at)
```

The interpretation for the metavariables, $MetaVar$, now carries the integer representing the metavariable itself but also carries information to identify whether this metavariable is supposed to be instantiated by a recursive member of the family or a opaque type. We do so by carrying a singleton [9] of type NA. This enables the compiler to gain knowledge over at when pattern-matching, which is important purely from the generic programming perspective. Without this information we would not be able to write a well-typed application function, for instance. We must know the types of the values supposed to be matched against a metavariable to ensure we will produce well-typed trees.

```
data MetaVar at = MetaVar Int (NA (Const Unit) at)
```

The type of changes over $Tree23$ can now be written using the generic representation for changes and metavariables.

```
type ChangeTree23 = Change Tree23Codes MetaVar (I 0)
```

We can read the type above as the type of changes over the *zero*-th (*I* 0) type within the mutually recursive family *Tree23Codes* with values of type *MetaVar* in its holes.

Computing Changes. Computing a *Change codes MetaVar* from a source and a destination elements of type *Fix codes ix* follows exactly the structure as we saw previously in Section 2: extract the pair of contexts and fix unbound metavariables in a postprocessing step. We are still assuming an efficient oracle *buildOracle s d* :: *Oracle codes*, that determines whether or not *an arbitrary t is a subtree of a fixed s and d indexed by n*, where:

```
type Oracle codes = ∀ j . Fix codes j → Maybe Int

buildOracle :: Fix codes i → Fix codes i → Oracle codes
```

The core of computing a change is in the extraction of the deletion and insertion contexts (*extract* function, Section 2). We must take care to avoid the problem we encountered in our previous implementation: a subtree that occurs in both the source and destination trees, but also occurs as the subtree of another common subtree (Figure 3) may result in unbound metavariables. We have shown how to fix this with a postprocessing step of the resulting change. That is still the case, but we now collect additional information from the context extraction before postprocessing.

Looking at the type of *Oracle*, we see it will only share recursive positions by construction. We use the *ForceI* type to bring this fact on the type level. That is, we are only sharing *recursive* positions so far. We also use the indexed product type (:*:) to carry along information.

```
data (:*:) f g x = f x :*: g x

data ForceI :: (Nat → ∗) → Atom → ∗ where
  ForceI :: f i → ForceI f (I i)
```

Defining the generic *txExtract* function is simple. We check whether a given *x* is a subtree of the source and destination trees by consulting the oracle. If so, we return a hole with the subtree annotated; if *x* is not a common subtree we extract the topmost constructor and recurse over its recursive positions.

```
txExtract :: Oracle codes
          → Fix codes ix
          → Tx codes (ForceI (Const Int :*: Fix codes)) (I ix)
txExtract ics x = case ics x of
  Just i  → TxHole (ForceI (Const i :*: x))
  Nothing → let Tag c p = sop (unFix x)
            in TxPeel c (mapNP (elimNA TxOpq (txExtract ics)) p)
```

Postprocessing works by traversing the result of the extracted contexts. In case we need to keep a given tree and forget that it was shared we convert it to a context with *na2tx*. Recall the reason for wanting to keep only the metavariables that occur in both the insertion and deletion contexts is to prevent any `undefined variable` when applying our patches, which would break correctness. More technically, the *txPostprocess* function groups the metavariables of each context in a set and computes the intersection of such sets, then maps over its arguments replacing the *Const Int* :*: *Fix codes* hole by either *Const Int*, if the *Int* belongs in the set, or by translating the *Fix codes* with $na2tx \circ NA_I$. The implementation is shown in Figure 7.

At this point, given two trees *a* and *b* of type *Fix codes ix*, we can extract and post-processed both the deletion and insertion contexts, of type *Tx codes* (*ForceI* (*Const Int*)) (*I ix*). These are just like a value of type *Fix codes ix* with holes of type *Const Int* exclusively in recursive positions. This is the generic version of the *changeTree23* function presented in Section 2:

```
txPostprocess :: Tx codes (ForceI (Const Int :*: Fix codes)) (I ix)
              → Tx codes (ForceI (Const Int :*: Fix codes)) (I ix)
              → Change (ForceI (Const Int)) (I ix)
txPostprocess del ins =
  -- We have to txJoin the results since keepOrDrop returns a Tx
  execState (Change <$> (txJoin <$> utxMapM keepOrDrop del)
                    <*> (txJoin <$> utxMapM keepOrDrop ins))
            (varsOf del `intersect` varsOf ins)
where
  -- traverses a Tx and puts all the variables in a Set.
  varsOf :: Tx codes (ForceI (Const Int :*: Fix codes)) (I ix) → Set Int
  -- check whether a variable is in both contexts and decides
  keepOrDrop (ForceI (Const mvar) :*: subtree) = do
    okvars ← get
    if var ∈ okvars then return (TxHole (ForceI (Const mvar)))
                    else return (na2tx (NA_I subtree))
```

Fig. 7. Post processing of contexts returning closed changes

```
change :: Fix codes ix → Fix codes ix → Change codes (ForceI (Const Int)) (I ix)
change x y = let ics = buildOracle x y
             in txPostprocess (txExtract ics x) (txExtract ics y)
```

Recall that this function will only produce closed changes. That is, a deletion and a insertion context that have the same set of variables. Intuitively, this means that every variable that is declared is used and vice-versa.

Minimizing the Changes: Computing Patches. The next step is to to minimize the changes coming from *change* function, yielding a *patch*. The generic counterpart of *Patch23* from Section 2 is given by:

```
type Patch codes at = Tx codes (Change codes MetaVar) at
```

Firstly, we have to identify the *spine*, that is, the constructors that are present in both the deletion and insertion contexts. This is done by splitting a big change into the *greatest common prefix* of the insertion and deletion contexts and the smaller changes inside:

```
txGCP :: Tx codes φ at → Tx codes ψ at → Tx codes (Tx codes φ :*: Tx codes ψ) at
txGCP (TxOpq x) (TxOpq y)
  | x ≡ y     = TxOpq x
  | otherwise = TxHole (TxOpq x :*: TxOpq y)
txGCP (TxPeel cx px) (TxPeel cy py)
  = case testEquality cx px of
      Nothing  → TxHole (TxPeel cx px :*: TxPeel cy py)
      Just Refl → TxPeel cx (mapNP (uncurry′ txGCP) (zipNP px py))
txGCP a b = TxHole (a :*: b)
```

The *txGCP* can is just like a generalized *zip* function, but instead of stopping whenever one of its arguments has been consumed and forgetting the other, it stores the rest of the other argument. It is *greatest* in the sense that it consumes as many constructors as possible and resorts to *TxHole* when it cannot progress further.

We know, from Section 2 that we cannot simply take the result of *change*, compute its *greatest common prefix* and be done with it. This would yield a patch with potentially malformed changes. The *txGCD* function is not aware of metavariables and might break their scoping.

Refining the result of *txGCP* is conceptually simple. All we have to do is bubble up the changes to a point where they are all *closed*, **??**. We start by creating some machinery to distinguish the open changes from the closed changes in the result of a *txGCP*. Then we define a traversal that shall look at those tags and decide whether more constructors should be pushed into the changes or not. This is done by the *closure* function.

Tagging open and closed changes is done with the indexed sum type. We use *InL* to mark the *open* changes and *InR* for the *closed* changes.

```
data Sum f g x = InL (f x) | InR (g x)

either' :: (f x → r x) → (g x → r x) → Sum f g x → r x
either' a b (InL fx) = a fx
either' a b (InR gx) = b gx
```

The *isClosed* predicate is responsible for deciding how to tag a change.

```
isClosed :: Change codes at → Sum (Change codes) (Change codes) at
isClosed (Change del ins)
  | variables ins ≡ variables del = InR (Change del ins)
  | otherwise                     = InL (Change del ins)
```

The *Sum* comes in handy for it can be passed as an argument to *Tx*, of kind *Atom* → ∗. This enables us to map our predicate over an arbitrary patch *p*:

```
txMap isClosed p :: Tx codes (Sum (Change codes) (Change codes)) at
```

The final *closure* function is defined with an auxiliary function, *closure'*. This auxiliary function receives a patch with tagged changes and tries to eliminate all the *open changes*, tagged with *InL*. We do so by finding the smallest closed change that binds the required variables. If the *closure'* function cannot find the constructor that binds all variables, it tags the patch as an *open change* altogether. The first three cases are trivial:

```
closure' :: Tx codes (Sum (Change codes) (Change codes)) at
         → Sum (Change codes) (Tx codes (Change codes)) at
closure' (TxOpq x) = InR (TxOpq x)
closure' (TxHole (InL oc)) = InL oc
closure' (TxHole (InR cc)) = InR cc
```

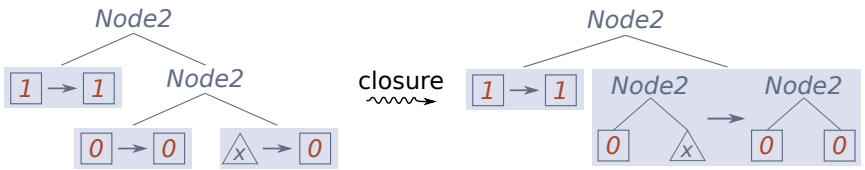
The interesting case of the *closure'* function is the *TxPeel* pattern, where we first try to compute the closures for the fields of the constructor and check whether all these fields contain only closed changes. If that is the case, we are done. If some fields contain open changes, however, the *mapNPM fromInR* fails with a *Nothing* and we must massage some data. This is akin to a simple distributive law for *Tx*, defined below.

```
distr :: Tx codes (Change codes φ) at → Change codes φ at
distr spine = Change (txJoin (txMap chgDel spine))
                     (txJoin (txMap chgIns spine))
```

The difference between *distr* and the *Nothing* clause in *closure′* is that in the later we are handling *NP* (*Tx codes* (*Change codes φ*)), i.e., a sequence of *Tx* instead of a single one. Consequently, we need some more code.

```
closure′ (TxPeel cx px)
  = let aux = mapNP closure′ px
    in case mapNPM fromInR aux of
      Just np  → InR (TxPeel cx np) -- everything is closed.
       -- some changes are open. Try pushing cx down the changes in px and
       -- see if this closes them, then.
      Nothing → let chgs = mapNP (either′ InL (InR ∘ distr)) aux
                    dels = mapNP (either′ chgDel chgDel) chgs
                    inss = mapNP (either′ chgIns chgIns) chgs
                    tmp = Change (TxPeel cx dels) (TxPeel cx inss)
                in if isClosed tmp
                  then InR (TxHole tmp)
                  else InL (Change tmp)
```

In the code above, *aux* is a sequence of either open changes or patches. The local *dels* and *inss* are defined as the a sequence deletion and insertion contexts from *aux*, regardless if they come from open or closed changes. This allows us to assemble a new, larger, change (*tmp*). Finally, we check whether this larger change is closed or not. We recall the illustration in Figure 5, repeated below, for a graphical intuition.



To finish it up, we wrap *closure′* within a larger function that always returns a *Tx* with all changes being *closed*. The necessary observation is that if we pass a given *tx* to *closure′* such that *distr tx* is closed, then *closure′* will always return a *InR* value. In our case, the *txPostprocess* is in place precisely to provided that guarantee, hence, the *error* is unreachable.

```
closure :: Tx codes (Sum (Change codes) (Change codes)) at
        → Patch codes at
closure = either′ (const $ error "no closure exists") id
```

The final *diff* function is then assembled by using the closure of the greatest common prefix of the change the comes from the *change* function. In order to further enlarge the domain of our patches we add a small additional step where we replace the opaque values in the spine with copies.

```
diff :: Fix codes ix → Fix codes ix → Patch codes (I ix)
diff x y = let Change del ins = change x y
           in closure $ txRefine TxHole mkCpy
                      $ txMap isClosed
                      $ txGCP del ins
```

The *txRefine* simply traverses the Tx and refines the holes and opaques into Txs using the provided functions. In our case we leave the holes unchanged and replace the occurrences of $TxOpq$ by a new *copy* change.

```
txRefine :: (∀ at . f at   → Tx codes g at)
         → (∀ k  . Opq k → Tx codes g (K k))
         → Tx codes f at → Tx codes g at
```

Applying Patches. Patch application follows closely the scheme sketched in for 2-3-trees (Section 2). There is one main difference, though. Our changes are now placed in the leaves of our spine and can be applied locally.

Our final *applyChange* will be responsible for receiving a change and a tree and instantiate the metavariables by matching the tree against the deletion context then substituting this valuation in the insertion context. Its type is given by:

```
applyChange :: Change codes MetaVar at
            → NA (Fix codes) at
            → Maybe (NA (Fix codes) at)
```

We are now left to match the spine with a value of $NA\ (Fix\ codes)$. and leave the changes paired up with the corresponding local elements they must be applied to. This is similar to the *txGCP*, and can be implemented by it. We must extract the greatest common prefix of the spine and the Tx that comes from translating the $NA\ (Fix\ codes)$ value but must make sure that the leaves have all $TxHole$s on the left.

```
txZipEl :: Tx codes φ at → NA (Fix codes) at → Maybe (Tx codes (φ :*: NA (Fix codes)))
txZipEl tx el = txMapM (uncurry' checkIsHole) $ txGCP tx (tx2na el)
  where
    checkIsHole :: Tx codes φ at → Tx codes ψ at → Maybe (φ :*: NA (Fix codes) at)
    checkIsHole (TxHole φ) el = (φ:*:) <$> na2tx el
    checkIsHole _          _  = Nothing
```

Finally, we define our application function. First we check whether the spine matches the element. If that is the case, we apply the changes, which are already paired with the parts of the element they must be applied to:

```
apply :: Patch codes ix → Fix codes ix → Maybe (Fix codes ix)
apply patch el = txZipEl patch el >>= return ∘ txMapM (uncurry' applyChange)
```

Whenever a patch p can be applied to an element x, that is, $apply\ p\ x$ returns $Just\ y$ for some y, we say that p is *applicable* to x.

4 DEFINING THE ORACLE

In the previous sections we have implemented a generic diffing algorithm assuming the existence of *an oracle*, that determines whether a given subtree should be copied or not. We have seen that the overall performance of our algorithm depends on answering that question efficiently: we perform

one query per constructor in the source and destination of the diff. In this section we provide an efficient construction for this oracle. Yet, it is worthwhile to define the inefficient, naive version first. Besides providing important intuition to what this function is doing it is an interesting generic programming exercise in its own right.

When deciding whether a given tree x is a subtree of two fixed trees s and d, a naive oracle would check every single subtree of s and d for equality against x. Upon finding a match, it would return the index of such subtree in the list of all subtrees. We enumerate all possible subtrees in a list with the help of *Exists* since the trees might have different indices.

```
data Exists :: (Atom n → *) → * where
  Ex :: f at → Exists f

subtrees :: Fix codes i → [Exists (Fix codes)]
subtrees x = Ex x : case sop x of
  Tag _ pt → concat (elimNP (elimNA (const [ ]) subtrees) pt)
```

Next, we define an equality over *Exist (Fix codes)* and search through the list of all possible subtrees for a match. The comparison function starts by comparing the indexes of the *Fix codes* values wrapped within *Ex* with *testEquality*. If they agree, we pattern match on *Refl*, which in turn allows us to compare the values for propositional equality.

```
heqFix :: Exists (Fix codes) → Exists (Fix codes) → Bool
heqFix (Ex x) (Ex y) = case testEquality x y of
  Nothing   → False
  Just Refl → x ≡ y
```

Finally, we put it all together in *buildOracle*: start by looking for our target, t, in the subtrees of x. Upon finding something, we proceed to check whether t also belongs in the subtrees of y. Since we are in the *Maybe* monad, if either of those steps fail, the entire function will return *Nothing*.

```
type Oracle codes = ∀ j . Fix codes j → Maybe Int

idx :: (a → Bool) → [a] → Maybe Int
idx f [ ] = Nothing
idx f (x : xs) = if f x then Just 0 else (1+) <$> idx f xs

buildOracle :: Fix codes i → Fix codes i → Oracle codes
buildOracle x y t = do
  ix ← idx (heqFix t) (subtrees x)
  iy ← idx (heqFix t) (subtrees y)
  return ix
```

There are two points of inefficiency this naive *buildOracle*. First, we build the *subtrees* list twice, once for the source and once for the destination. This is inherent to this approach and cannot be avoided. We then proceed to compare a third tree, t, for equality with every subtree in the lists of subtrees. The performance of this operation can be significantly improved.

In order to compare trees for equality in constant time we can annotate them with cryptographic hashes [18] and compare these hashes instead. This technique transforms our trees into *merkle trees* [19] and is more commonly seen in the security and authentication context [20, 22]. The generic programming machinery that is already at our disposal enables us to create *merkle trees* generically quite easily. The `generics-mrsop` provide some attribute grammar [16] functionality, in particular the computation of synthesized attributes arising from a fold. The *synthesize* function is just like a catamorphism, but we decorate the tree with the intermediate results at each node,

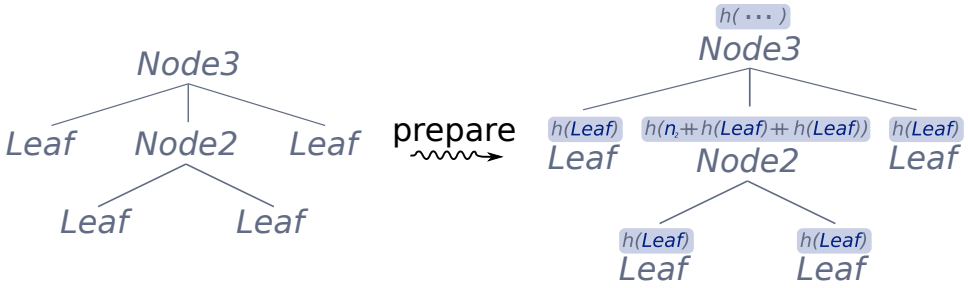


Fig. 8. Example of a merkelized *Tree23*, where n_2 is some fixed identifier and h is a hash function.

rather than only using them to compute the final outcome. This enables us to decorate each node of a *Fix codes* with a unique identifier (as shown in Figure 8) by running the *prepare* function, defined below.

```
newtype AnnFix x codes i = AnnFix (x i, Rep (AnnFix x codes) (Lkup i codes))

prepare :: Fix codes i → AnnFix (Const Digest) codes i
prepare = synthesize authAlgebra
```

Here, *AnnFix* is the cofree comonad, used to add a label to each recursive branch of our generic trees. In our case, this label will be the cryptographic hash of the concatenation of its subtree's hashes. Figure 8 shows an example of an input and corresponding output of the *prepare* function, producing a *merkelized Tree23*. The *synthesize* generic combinator annotates each node of the tree with the result of the catamorphism called at that point with the given algebra. Our algebra is sketched in pseudo-Haskell below:

```
authAlgebra :: Rep (Const Digest) sum → Const Digest iy
authAlgebra rep = case sop rep of
  Tag c [p₁, ..., pₙ] → Const ∘ sha256Concat
                        $ [encode c, encode (getSNat @iy), p₁, ..., pₙ]
```

We must append the index of the type in question, in this case *getSNat @iy*, to our hash computation to differentiate constructors of different types in the family represented by the same number.

Once we have a tree fully annotated with the hashes for its subtrees, we store them in a search-efficient structure. Given that a hash is just a [*Word*], the optimal choice is a Trie [6] mapping a *Word* to *Int*, where the *Int* is the value of the *metavariable* that will be assigned to the tree, in case it is a common subtree Looking up whether a tree x is a subtree of some tree s can be done by looking up x's topmost hash, also called the *merkle root*, against the trie generated from s. This is a very fast operation and hardly depends on the number of elements in the trie. In fact, this lookup runs in amortized constant time.

In this situation, however, we have to determine whether or not a tree x occurs as a subtree of *both* the source and destination trees. This is no harder, as we can efficiently compute the intersection of the tries arising from the source and target trees. Querying this trie will tell us whether or not a x occurs as a subtree of both trees.

It is of paramount importance to avoid recomputing the merkle root of a tree x each time we wish to know whether it is a common subtree. Otherwise, we still end up with an exponential algorithm. The solution is quite simple: we use *AnnFix (Const Digest) codes* in the *txExtract* function and the

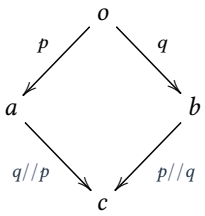


Fig. 9. Merge square

type of our oracle, where *Fix codes* was used before. This provides access to the merkle root in constant time. After this small modification to our *Oracle*, allowing it to receive trees annotated with hashes we proceed to define the efficient *buildOracle* function.

```
type Oracle codes = ∀ j . AnnFix (Const Digest) codes j → Maybe Int

buildOracle :: Fix codes i → Fix codes i → Oracle codes
buildOracle s d = let s' = prepare s
                      d' = prepare d
  in lookup (mkSharingTrie s' `intersect` mkSharingTrie d')
  where
    -- insert all digests in a trie
    mkSharingTrie :: AnnFix (Const Digest) codes j → Trie Word Int
```

Where the *mkSharingTrie* function will maintain a counter and traverse its argument. At every node it will insert an entry with that node's hash and the counter value. It then increases the counter and recurses over the children. The same subtree might appear in different places in s and d, for the *Int* associated with it will differ from *mkSharingTrie* s' and *mkSharingTrie* d'. This is not an issue if we take an *intersect* function with type $Trie\ k\ v \to Trie\ k\ t \to Trie\ k\ v$, hence, keeping only the assignments from the first trie such that the key is also an element of the second.

We can easily get around hash collisions by computing an intermediate *Trie* from *Word* to *Exists (Fix codes)* in the *mkSharingTrie* function and every time we find a potential collision we check the trees for equality. If equality check fails, a hash collision is found and the entry would be removed from the map. When using a cryptographic hash, the chance of collision is negligible and we chose to ignore it.

5 MERGING PATCHES

One of the main motivations for generic structure-aware diffing is being able to merge patches in a more structured fashion than using `diff3`, which considers changes to every line. In the past, structural merging has proven to be a difficult task [23, 30] even for the easiest cases. This section shows how our new structure for representing changes enables us to write a simple merge algorithm, offering both acceptable performance and a improvement over `diff3`. We will sketch the implementation of our algorithm here and evaluate its performance in Section 6.

The merging problem, illustrated in Figure 9, is the problem of computing a new patch, $q \,//\, p$, given two patches p and q. It consists in a patch that contains the changes of q potentially adapted to work on a value that has already been modified by p. This is sometimes called the *transport* of q over p or the *residual* [12] of p and q.

There is a class of patches that are trivial to merge: those that modify separate locations of a tree. If p and q are disjoint, then $p \mathbin{//} q$ can return p without further adaptations. Our algorithm shall merge only disjoint patches, marking all other situations as a conflict. We choose to represent conflicts as a pair of overlapping patches.

```
type Conflict codes = Patch codes :*: Patch codes

type PatchConf codes = Tx codes (Sum (Conflict codes) (Change codes))
```

In practice, it may be desirable to record further meta-information to facilitate conflict resolution.

Our merging operator, $(//)$, receives two patches and returns a patch possibly annotated with conflicts. We do so by matching the spines, and carefully inspecting any changes where the spines differ.

```
(//) :: Patch codes at → Patch codes at → PatchConf codes at
```

The intuition here is that $p \mathbin{//} q$ must preserve the intersection of the spines of p and q and reconcile the differences whenever one of the patches has a change. Note that it is impossible to have completely disjoint spines since p and q are applicable to at least one common element. Using the *greatest common prefix* function defined previously, we can zip together the shared spines, pushing the resolution down to the leaves:

```
p // q = txMap (uncurry' reconcile) $ txGCP p q
```

Here, the *reconcile* function shall check whether the disagreeing parts are disjoint, i.e., either one of the two changes is the identity or they perform the exactly same change. If that is the case, the *reconcile* function returns its first argument. In fact, this is very much in line with the properties of a residual operator [12].

```
reconcile :: Patch codes at → Patch codes at → Sum (Conflict codes) (Change codes) at
reconcile p q
  | patchIden p ∨ patchIden q = InR $ distr p
  | patchEquiv p q            = InR $ copy
  | otherwise                 = InL $ Conflict p q
```

We see the code for *reconcile* closely follows the definition of disjointness above—one of the patches must be the identity or they are equal. The *patchIden* functions checks whether all changes in that patch are copies and *patchEquiv* checks if two patches are α-equivalent. Taking a closer look at the *reconcile* function, we see it follows the three identity laws from residual theory. The first branch agrees from the two identity laws from residual theory that state that $p \mathbin{//} id \equiv p$ and $id \mathbin{//} p \equiv id$, whereas the second branch follows the third identity law, which states that $p \mathbin{//} p \equiv id$, meaning that applying a patch over something that has been modified by this very patch amounts to not changing anything.

Our trivial merge algorithm returns a conflict for non-disjoint patches, but this does not mean that it is impossible to merge them in general. Although a full discussion is out of the scope of this paper, there are a number of non-disjoint patches that can still be merged. These non-trivial merges can be divided in two main situations: (A) there is no action needed even though patches are not disjoint, and (B) the relevant parts of a patch can be transported to operate on different parts tree automatically. In Figure 10 we illustrate situations (A) and (B) in the merge square for two non-disjoint patches. In the top subfigure we see the residual returning the patch unaltered (case A). In this example, the patch in the 'nominator' position is a simple swap of subtrees. This swap operation can be applied to every possible result of applying the second patch in the 'denominator' of the residual. As a result, it computing the residual is easy: we simply return the 'nominator'

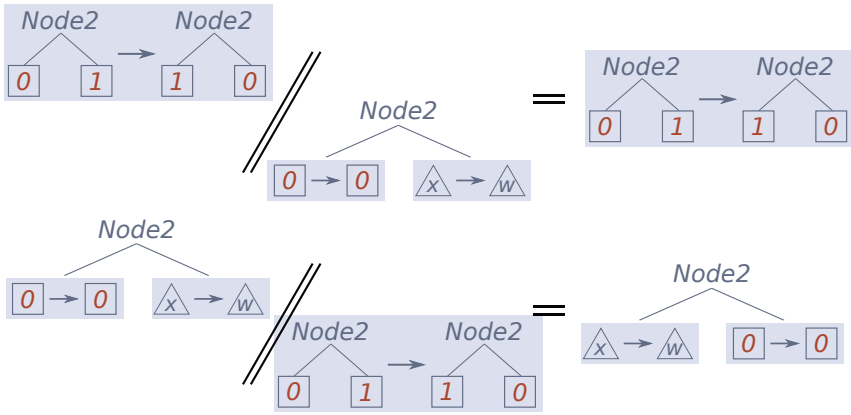


Fig. 10. Example of a merge square where the first residual is obtained by not changing the patch and the second is computed by applying a patch to another patch, transporting the changes.

patch. In the second subfigure, however, the situation is reversed. The 'denominator' patch must be applied to the 'nominator' – yielding a new patch that has the expected behavior. In future work, we hope to identify the precise conditions under which two non-disjoint patches can be merged in this way.

6 EXPERIMENTS

We have conducted two experiments over a number of Lua [14] source files. We obtained these files data by mining the top Lua repositories on GitHub and extracting all the merge conflicts recorded in their history. Next, we ran two experiments over this data: a performance test and a merging test. We chose the Lua programming language for two reasons. First, there is a Haskell parser for Lua readily available on Hackage and, secondly, due to a performance bug [11] in GHC we are not able to instantiate our generic algorithm to more complex abstract syntax trees, such as that of C.

Performance Evaluation. In order to evaluate the performance of our implementation we have timed the computation of the two diffs, $diff\ o\ a$ and $diff\ o\ b$, for each merge conflict a, o, b in our dataset. In order to ensure that the numbers we obtained are valid and representative of a real execution trace we timed the execution time of parsing the files and running $length \circ encode \circ uncurry\ diff$ over the parsed files, where *encode* comes from *Data.Serialize*. Besides ensuring that the patch is fully evaluated, the serialization also mimics what would happen in a real version control system since the patch would have to be saved to disk. After timing approximately 1200 executions from real examples we have plotted the data over the total number of constructors for each source-destination pair. In Figure 12 we see two plots: on the left we have plotted 70% of our dataset in more detail whereas on the right we show the full plot. The results were expected given that we seen how $diff\ x\ y$ runs in $O(n+m)$ where n and m are the number of constructors in x and y abstract syntax trees, respectively. Confirming our analysis with empirical further strengthens our algorithm as a practical implementation of structured differencing.

Merging Evaluation. We have also performed a preliminary evaluation of the simple merging algorithm presented in Section 5. After collecting all the conflicts from the GitHub repositories, we attempted to merge the structured diffs that our algorithm computes. When this merge succeeded, we checked that the resulting merge square (Figure 9) commutes as expected. In this way, we

Repository	Commits	Contributors	Total Conflicts	Trivial Conflicts
awesome	9289	250	5	0
busted	936	39	9	0
CorsixTH	3207	64	25	8
hawkthorne-journey	5538	61	158	27
kong	4669	142	163	11
koreader	6418	89	14	2
luakit	4160	62	28	2
luarocks	2131	51	45	3
luvit	2862	68	4	1
nn	1839	177	3	0
Penlight	730	46	6	3
rnn	622	42	6	1
snabb	8075	63	94	6
tarantool	12299	82	33	2
telegram-bot	729	50	5	0
		total	598	66

Fig. 11. Lua repositories mined from *GitHub*

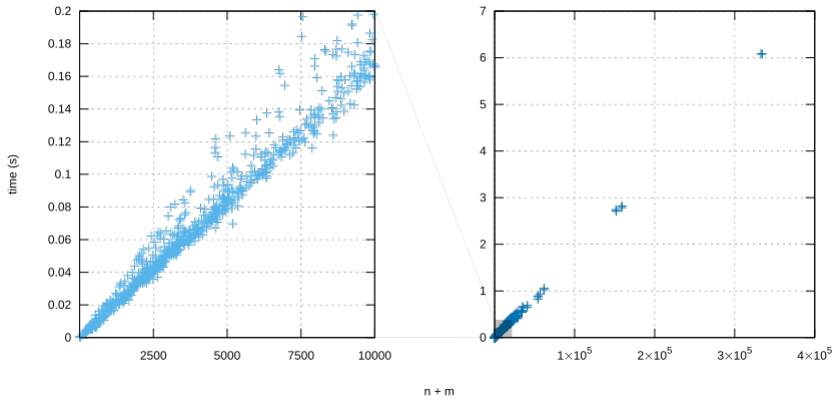


Fig. 12. Plot of the time for diffing two lua files over the total AST nodes

were able to solve a total of 66 conflicts automatically, amounting to 11% of all the conflicts we encountered. We consider these initial numbers to be encouraging: even a naive merge algorithm on structured changes manages to outperform the current state of the art. We expect that a more refined notion of merging may improve these results further.

Threats to Validity

There are two main threats to the validity of our empirical results. Firstly, we are diffing and merging *abstract* syntax trees, hence ignoring comments and formatting. There would be no extra

effort in handling these issues, beyond recording them explicitly in a more concrete syntax tree and adapting our parser to produce such trees. Nevertheless, one might expect expect a slightly lower success rate since we are ignoring formatting changes altogether. Secondly, a significant number of developers prefer to rebase their branches instead of merging them. Therefore, we may have missed a number of important merge conflicts that are no longer recorded, as rebasing erases history. Our merge algorithm might be able to resolve some of these conflicts automatically—but there is no way to establish this.

7 DISCUSSION AND CONCLUSION

The results from Section 6 are very encouraging. We see that our diffing algorithm has competitive performance and our trivial merging operation is capable of merging changes where `diff3` fails. Yet there is still plenty of work to be done.

Future Work

Controlling Sharing. One interesting direction for further work is how to control the sharing of subtrees. As it stands, the differencing algorithm will share every subtree that occurs in both the source and destination files. This can lead to undesirable behavior. For example, we may not want to share *all* occurrences of a variable within a program, but rather only share occurrences of a variable with the same binder. That is, sharing should respect the scope variables. A similar question arises with constants – should all occurrences of the number 1 be shared?

There are a variety of options to customize the sharing behavior of our algorithm. One way to do so would allow the definition of a custom oracle that is scope-aware. By hashing both the identifier name and its binder, we can ensure that variables are not shared over scope boundaries. Another option would be consider abstract syntax trees that make the binding structure of variables explicit.

Better Merge Algorithm. The merging algorithm presented in Section 5 only handles trivial cases. Being able to merge patches that are not disjoint is the subject of ongoing research. The problem seems related to unification, residual systems, and rewriting systems. We hope that relating the merging problem to these settings might help nail down the necessary conditions for merging to succeed. One would expect that it would have some resemblance to a pushout, as in pointed out by Mimram and Di Giusto [21].

Automatic Merge Strategies. We would like to develop a language to specify domain specific strategies for conflict resolution. For instance, whenever the merging tool finds a conflict in the `build-depends` section of a cabal file, it might try sorting the packages alphabetically and keeping the conflicting packages with the higher version number. Ideally, these rules should be simple to write, yet still allow a high degree of customization.

Formalization and Meta-theory. We would be happy to engage in a formalization of our work with the help of a proof assistant. This would help to develop the meta-theory and provide definite confidence that our algorithms are correct with respect to their specification. This could be achieved by rewriting our code in Agda [25] whilst proving the correctness properties we desire. This process would provide invaluable insight into developing the meta-theory of our system.

Extending the Generic Universe. Our prototype is built on top of `generics-mrsop`, a generic programming library for handling mutually recursive families in the sums of products style. With recent advances in generic programming [27], we might be able to extend our algorithm to handle mutually recursive families that have `GADTs`.

Related Work

Related work can be classified in the treatment of types. The untyped tree differencing problem was introduced in 1979 [29] as a generalization of the longest common subsequence problem [4]. There has been a significant body of work on the untyped tree differencing problem [1, 8, 15], but these results do not transport to the typed setting: the transformations that are computed are not guaranteed to produce well-typed trees.

The first datatype generic algorithm was presented by Lempsink and Löh [17], which was later extended by Vassena [30]. Their work consists largely in using the same algorithm as `diff` on the flattened representation of a tree. The main observation is that basic operations (insertion, deletion and copy) can be shown to be well-typed when operating on these flattened representations. Although one could compute differences with reasonably fast algorithms, merging these changes is fairly difficult and in some cases might be impossible [30]. Miraldo et al. [23] take a slightly different approach, defining operations that work directly on tree shaped data. Using this approach, changes become easier to merge but harder to compute. Both bodies of work follow the same general idea as the untyped variants: compute all possible patches and select the 'best' patch from these alternatives. As we have already mentioned (Section 1), this is not an optimal strategy. The number of patches grows explosively and defining the *best* patch using insertions, deletions and copies is impossible without further heuristics .

The work of Asenov et al. [3] is also untyped, but uses a different technique for finding the diff: it flattens trees and embellishes the resulting lists with additional annotations, and then uses the UNIX `diff` tool to compute patches. Finally, it transports the changes back to the tree-shaped datatypes using the annotations that were added. The authors identify a number of interesting situations that occur when merging tree differences. The `gumtree` [10] project, explores a similar line of work, but uses its own algorithm for computing graph transformations between untyped representations of abstract syntax trees.

There have been several different approaches to formalizing a theory of patches. The version control system `darcs` [26] was one of the first to present a more formal theory of patches, but the patches themselves were still line-based. Mimram and De Giusto [21] have developed a theoretical model of line-based patches in a categorical fashion. This has inspired the version control system `pijul`. Swierstra and Löh [28] have proposed using separation logic to define a meta-theory of patches and merging. Finally, Angiuli et al. [2] describe a patch theory based on homotopy type theory.

Conclusions

Throughout this paper we have developed an efficient type-directed algorithm for computing structured differences for a large class of algebraic datatypes, namely, mutually recursive families. This class of types can represent the abstract syntax tree of most programming languages and, hence, our algorithm can be readily instantiated to compute the difference between programs written in these languages. We have validated our implementation by computing diffs between Lua [14] source files obtained from various repositories on GitHub; the algorithm's run-time is competitive, and even a naive merging algorithm already offers a substantial improvement over existing technology. Together, these results demonstrate both a promising direction for further research and a novel application of the generic programming technology that is readily available in today's functional languages.

REFERENCES

[1] Tatsuya Akutsu, Daiji Fukagawa, and Atsuhiro Takasu. 2010. Approximating Tree Edit Distance through String Edit Distance. *Algorithmica* 57, 2 (2010), 325–348. https://doi.org/10.1007/s00453-008-9213-z

[2] Carlo Angiuli, Edward Morehouse, Daniel R. Licata, and Robert Harper. 2014. Homotopical Patch Theory. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming (ICFP '14)*. ACM, New York, NY, USA, 243–256. https://doi.org/10.1145/2628136.2628158

[3] Dimitar Asenov, Balz Guenat, Peter Müller, and Martin Otth. 2017. Precise Version Control of Trees with Line-Based Version Control Systems. In *Proceedings of the 20th International Conference on Fundamental Approaches to Software Engineering - Volume 10202*. Springer-Verlag New York, Inc., New York, NY, USA, 152–169. https://doi.org/10.1007/978-3-662-54494-5_9

[4] L. Bergroth, H. Hakonen, and T. Raita. 2000. A survey of longest common subsequence algorithms. In *String Processing and Information Retrieval, 2000. SPIRE 2000. Proceedings. Seventh International Symposium on*. 39–48.

[5] Philip Bille. 2005. A survey on tree edit distance and related problems. *Theor. Comput. Sci* 337 (2005), 217–239.

[6] Peter Brass. 2008. *Advanced Data Structures* (1 ed.). Cambridge University Press, New York, NY, USA.

[7] Edsko de Vries and Andres Löh. 2014. True Sums of Products. In *Proceedings of the 10th ACM SIGPLAN Workshop on Generic Programming (WGP '14)*. ACM, New York, NY, USA, 83–94. https://doi.org/10.1145/2633628.2633634

[8] Erik D. Demaine, Shay Mozes, Benjamin Rossman, and Oren Weimann. 2007. An Optimal Decomposition Algorithm for Tree Edit Distance. In *Proceedings of the 34th International Colloquium on Automata, Languages and Programming (ICALP 2007)*. Wroclaw, Poland, 146–157.

[9] Richard A. Eisenberg and Stephanie Weirich. 2012. Dependently Typed Programming with Singletons. In *Proceedings of the 2012 Haskell Symposium (Haskell '12)*. ACM, New York, NY, USA, 117–130. https://doi.org/10.1145/2364506.2364522

[10] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-grained and accurate source code differencing. In *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*. 313–324. https://doi.org/10.1145/2642937.2642982

[11] GHC Trac. 2018. Memory usage exploding for complex pattern matching. (2018). https://ghc.haskell.org/trac/ghc/ticket/14987#no2.

[12] Gérard Huet. 1994. Residual theory in λ-calculus: a formal development. *Journal of Functional Programming* 4, 3 (1994), 371âĂŞ394. https://doi.org/10.1017/S0956796800001106

[13] J. W. Hunt and M. D. McIlroy. 1976. *An Algorithm for Differential File Comparison.* Technical Report CSTR 41. Bell Laboratories, Murray Hill, NJ.

[14] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes Filho. 1996. LuaâĂŤAn Extensible Extension Language. *Software: Practice and Experience* 26, 6 (1996), 635–652. https://doi.org/10.1002/(SICI)1097-024X(199606)26:6<635::AID-SPE26>3.0.CO;2-P

[15] Philip N. Klein. 1998. Computing the Edit-Distance Between Unrooted Ordered Trees. In *Proceedings of the 6th Annual European Symposium on Algorithms (ESA '98)*. Springer-Verlag, London, UK, UK, 91–102.

[16] Donald E. Knuth. 1990. The Genesis of Attribute Grammars. In *Proceedings of the International Conference WAGA on Attribute Grammars and Their Applications*. Springer-Verlag, London, UK, UK, 1–12. http://dl.acm.org/citation.cfm?id=645938.671208

[17] Eelco Lempsink, Sean Leather, and Andres Löh. 2009. Type-safe Diff for Families of Datatypes. In *Proceedings of the 2009 ACM SIGPLAN Workshop on Generic Programming (WGP '09)*. ACM, New York, NY, USA, 61–72.

[18] Paul van Oorschot Menezes A. J. and Scott A. Vanstone. [n. d.]. *Handbook of Applied Cryptography* (boca raton, xiii, 780, 1997 ed.). CRC Press.

[19] Ralph C. Merkle. 1988. A Digital Signature Based on a Conventional Encryption Function. In *Advances in Cryptology — CRYPTO '87*, Carl Pomerance (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 369–378.

[20] Andrew Miller, Michael Hicks, Jonathan Katz, and Elaine Shi. 2014. Authenticated Data Structures, Generically. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '14)*. ACM, New York, NY, USA, 411–423. https://doi.org/10.1145/2535838.2535851

[21] Samuel Mimram and Cinzia Di Giusto. 2013. A Categorical Theory of Patches. *CoRR* abs/1311.3903 (2013). arXiv:1311.3903 http://arxiv.org/abs/1311.3903

[22] Victor Cacciari Miraldo, Harold Carr, Alex Kogan, Mark Moir, and Maurice Herlihy. 2018. Authenticated Modular Maps in Haskell. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Type-Driven Development (TyDe 2018)*. ACM, New York, NY, USA, 1–13. https://doi.org/10.1145/3240719.3241790

[23] Victor Cacciari Miraldo, Pierre-Évariste Dagand, and Wouter Swierstra. 2017. Type-directed diffing of structured data. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Type-Driven Development*. ACM, 2–15.

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

[25] Ulf Norell. 2009. Dependently Typed Programming in Agda. In *Proceedings of the 4th International Workshop on Types in Language Design and Implementation (TLDI '09)*. ACM, New York, NY, USA, 1–2.

[26] David Roundy. 2005. Darcs: Distributed Version Management in Haskell. In *Proceedings of the 2005 ACM SIGPLAN Workshop on Haskell (Haskell '05)*. ACM, New York, NY, USA, 1–4. https://doi.org/10.1145/1088348.1088349

[27] Alejandro Serrano and Victor Cacciari Miraldo. 2018. Generic Programming of All Kinds. In *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell (Haskell 2018)*. ACM, New York, NY, USA, 41–54. https://doi.org/10.1145/3242744.3242745

[28] Wouter Swierstra and Andres Löh. 2014. The Semantics of Version Control. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward! '14)*. 43–54.

[29] Kuo-Chung Tai. 1979. The Tree-to-Tree Correction Problem. *J. ACM* 26, 3 (July 1979), 422–433. https://doi.org/10.1145/322139.322143

[30] Marco Vassena. 2016. Generic Diff3 for Algebraic Datatypes. In *Proceedings of the 1st International Workshop on Type-Driven Development (TyDe 2016)*. ACM, New York, NY, USA, 62–71.

[31] Brent A. Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. 2012. Giving Haskell a Promotion. In *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI '12)*. ACM, New York, NY, USA, 53–66. https://doi.org/10.1145/2103786.2103795