# Best Title in the Universe

42

Victor Cacciari Miraldo  Wouter Swierstra
University of Utrecht
{v.cacciarimiraldo,w.s.swierstra} at uu.nl

## Abstract

stuff

***Categories and Subject Descriptors*** D.1.1 [*look*]: for—this

***General Terms*** Haskell

***Keywords*** Haskell

## 1. Introduction

The majority of version control systems handle patches in a non-structured way. They see a file as a list of lines that can be inserted, deleted or modified, with no regard to the semantics of that specific file. The immediate consequence of such design decision is that we, humans, have to solve a large number of conflicts that arise from, in fact, non conflicting edits. Implementing a tool that knows the semantics of any file we happen to need, however, is no simple task, specially given the plethora of file formats we see nowadays.

This can be seen from a simple example. Lets imagine Alice and Bob are iterating over a cake's recipe. They decide to use a version control system and an online repository to keep track of their modifications.

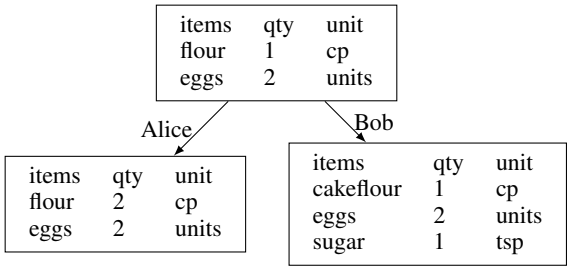


**Figure 1.** Sample CSV files

Lets say that both Bob and Alice are happy with their independent changes and want to make a final recipe. The standard way to track differences between files is the `diff3` [FIXBIB] unis tool. Running `diff3 Alice.csv 0.csv Bob.csv` would result in the output presented in figure 2. Every tag `====` marks a difference. Three locations follows, formatted as `file:line type`. The change type can be a *Change*, *Append* or *Delete*. The first one, says that file 1 (`Alice.csv`) has a change in line 2 (`1:2c`) which is `flour, 2 , cp`; and files 2 and 3 have different changes in the same line. The tag `====3` indicates that there is a difference in file 3 only. Files 1 and 2 should append what changed in file 3 (line 4).

```
====
1:2c
  flour, 2  , cp
2:2c
  flour, 1  , cp
3:2c
  cakeflour, 1  , cp
====3
1:3a
2:3a
3:4c
  sugar, 1  , tsp
```

**Figure 2.** Output from `diff3`

If we try to merge the changes, `diff3` will flag a conflict and therefore require human interaction to solve it, as we can see by the presence of the `====` indicator in its output. However, Alice's and Bob's edits, in figure 1 do *not* conflict, if we take into account the semantics of CSV files. Although there is an overlapping edit at line 1, the fundamental editing unit is the cell, not the line.

We propose a structural diff that is not only generic but also able to track changes in a way that the user has the freedom to decide which is the fundamental editing unit. Our work was inspired by [**?** ] and [**?** ]. We did extensive changes in order to handle structural merging of patches. We also propose extensions to this algorithm capable of detecting purely structural operations such as refactorings and cloning.

The paper begins by exploring the problem, generically, in the Agda [FIXBIB] language. Once we have a provably correct algorithm, the details of a Haskell implementation of generic diff'ing are sketched. To open ground for future work, we present a few extensions to our initial algorithm that could be able to detect semantical operations such as *cloning* and *swapping*.

### Contributions

- *Study of a more algebraic patch theory.*
- *Agda model.*
- *Haskell Prototype.*

### Background

- *Should we have this section? It cold be nice to at least mention the edit distance problem and that in the untyped scenario, the best running time is of $O(n^3)$. Types should allow us to bring this time lower.*

## 2. Structural Diffing

Alice and Bob were both editing a CSV file which represents data that is isomorphic to $[[Atom\ String]]$, where $Atom\ a$ is a simple tag that indicates that $a$s should be treated as atomic.

- *What do we mean by structural?*
- *Give some context: Tree-edit distance;*
- *We seek to obtain a system with something close to residuals.*

**To Research!**

- *Check out the antidiagonal with more attention: `http://blog.sigfpe.com/2007/09/type-of-distinct-pairs.html`*
- *The LCS problem is closely related to diffing. We want to preserve the LCS of two structures! How does our diffing relate? Does this imply maximum sharing?*

### 2.1 Context Free Datatypes

- *Explain the universe we're using.*
- *Explain the intuition behing our $D$ datatype.*
- *Mention that it is correct.*

Took from [**?** ].

```
data U : ℕ → Set where
  u0  : {n : ℕ} → U n
  u1  : {n : ℕ} → U n
  _⊕_ : {n : ℕ} → U n → U n → U n
  _⊗_ : {n : ℕ} → U n → U n → U n
  β   : {n : ℕ} → U (suc n) → U n → U n
  μ   : {n : ℕ} → U (suc n) → U n
  vl  : {n : ℕ} → U (suc n)
  wk  : {n : ℕ} → U n → U (suc n)
```

- *Explain the patching problem.*
- *We want a type-safe approach.*
- *Argue that the types resulting from our parser are in a sub-language of what we treated next.*
- *introduce* edit-script, diffing *and* patching *or* apply

### 2.2 Patches over a Context Free Type

- *Explain that a patch is something which we can apply.*
- *Loh's approach is too generic, as the diff function should have type $a \to a \to D\ a$.*

In order to simplify the presentation, we are gonna explicitely name variables and write our types in a more mathematical fashion, other than the Agda encoding. As we discussed earlier, a patch is an object that track differences in a given type. Different types will allow for different types of changes.

**Definition 2.1** (Simple Patch)**.** *We define a (simple) patch $D\ ty$ by induction on $ty$ as:*

$$\begin{aligned} D\ 0 &= 0 \\ D\ 1 &= 1 \\ D\ (x \times y) &= D\ x \times D\ y \\ D\ (x + y) &= (D\ x + D\ y) + 2 \times (x \times y) \\ D\ (\mu X.F\ X) &= \mu X.(1 \\ &+ D\ (F\ 1) \times X \\ &+ 2 \times (F\ 1) \times X \\ &) \end{aligned}$$

*Where 1 and 0 are the usual terminal and initial objects of a given category.*

Let's see the coproduct case in more detail. There are four different possibilities for the changes seen in a coproduct, just like there are four different combinations of constructors for two objects of type $Either\ a\ b$. The first and second options, namelly $D\ x$ and $D\ y$ track differences of a $Left\ a$ into a $Left\ a'$ and a $Right\ b$ into a $Right\ b'$, respectively. The other possibilities are representing a $Left\ a$ becoming a $Right\ b$ or vice-versa. The other branches are straight-forward.

***Producing Patches*** Definition 2.1 provides us with some intuition on how one would define patches for a given datatype. The actual definition (figure 4) is more complicated, though. The Diff type takes one parameter, used to give a free-monad [FIXBIB] structure, and two indexes which indicate the type for which that diff is intented. We then define:

```
Patch : {n : ℕ} → Tel n → U n → Set
Patch t ty = D ⊥p t ty
```

Where $\bot_p = \lambda\_\ \_ \to \bot$.

Our first goal is to produce patches, or to differentiate between to objects of the same type. We can do that generically through

```
gdiff : {n : ℕ}{t : Tel n}{ty : U n}
  → ElU ty t → ElU ty t → Patch t ty
gdiff {ty = vl} (top a) (top b)      = D-top (gdiff a b)

gdiff {ty = wk u} (pop a) (pop b)  = D-pop (gdiff a b)

gdiff {ty = β F x} (red a) (red b) = D-β (gdiff a b)

gdiff {ty = u1} void void = D-void

gdiff {ty = ty ⊗ tv} (ay , av) (by , bv)
  = D-pair (gdiff ay by) (gdiff av bv)

gdiff {ty = ty ⊕ tv} (inl ay) (inl by) = D-inl (gdiff ay by)
gdiff {ty = ty ⊕ tv} (inr av) (inr bv) = D-inr (gdiff av bv)
gdiff {ty = ty ⊕ tv} (inl ay) (inr bv) = D-setl ay bv
gdiff {ty = ty ⊕ tv} (inr av) (inl by) = D-setr av by

gdiff {ty = μ ty} a b = D-mu (gdiffL (a :: []) (b :: []))
```

- *wrap it up?*

**Definition 2.2** (Defined)**.** *We say that a patch $p_a$ is defined for an input $a$ iff there exists an object $a'$ such that:*

$$apply\ p_a\ a \equiv Just\ a'$$

***Fixed Points*** The treatment for fixed points has to be made uniform, somehow, if we want a generic algorithm by the end of the day. What makes fixed points different than regular algebraic types is that they can grow or shrink arbitralily, and our diff function has to take that into account.

Recalling the fixed point clause of simple patches (def 2.1),

$$\begin{aligned} D\ (\mu X.F\ X) &= \mu X.(1 \\ &+ D\ (F\ 1) \times X \\ &+ 2 \times (F\ 1) \times X \\ &) \end{aligned}$$

it is straight forward to see that the $D\ (\mu X.F\ X)$ is isomorphic to a list with three recursive constructors and a non-recursive one. Following the edit operations studied by Löh[**?** ], we have an *insert*, a *delete* and a *end* edit operations. The big difference is that instead of copying, we have a constructor that track changes inside a constructor of $\mu X.F\ X$, we call this a *down* edit operation.

We heavily rely on the facth that $\mu X.F\ X \approx F\ 1 \times [\mu X.F\ X]$, that is, any inhabitant of a fixed-point type can be seen as a non-recursive head and a list of recursive children, or, expressed in our generic setting:

```
Openμ : {n : ℕ} → Tel n → U (suc n) → Set
Openμ t ty = ElU ty (tcons u1 t) × List (ElU (μ ty) t)

μ-open : {n : ℕ}{t : Tel n}{ty : U (suc n)}
  → ElU (μ ty) t → Openμ t ty

μ-close : {n : ℕ}{t : Tel n}{ty : U (suc n)}
  → Openμ t ty → Maybe (ElU (μ ty) t × List (ElU (μ ty) t))
```

Although we could have used vectors of a fixed length and made this a total isomorphism, we would have more problems than benefits. This will be discussed in section 2.5. Nonetheless, an important soundness result has been proven:

```
μ-close-resp-arity
  : {n : ℕ}{t : Tel n}{ty : U (suc n)}{a : ElU (μ ty) t}
    {hdA : ElU ty (tcons u1 t)}{chA l : List (ElU (μ ty) t)}
  → μ-open a ≡ (hdA , chA)
  → μ-close (hdA , chA ++ l) ≡ just (a , l)
```

### 2.3 The Cost Function

### 2.4 Sharing of Recursive Subterms

- If we want to be able to share recursive subexpressions we need a mutually recursive approach.
- Or, this will be handled during conflict solving. See refactoring.

### 2.5 Remarks on Type Safety

- only the interface to the user can be type-safe, otherwise we don't have our free-monad multiplication.

## 3. Patch Propagation

Let's say Bob and Alice perform edits in a given object, which are captured by patches $p$ and $q$, shown in figure 3. The natural question to ask is *how do we join these changes*.

The residual $p/q$ of two patches $p$ and $q$ only makes sense if both $p$ and $q$ are aligned, that is, are defined for the same input. It captures the notion of incorporating the changes made by $p$ in an object that has already been modified by $q$.

- *Pijul has this notion of handling a merge as a pushout, but it uses the free co-completion of a rather simple category. This doesn't give enough information for structured conflict solving.*

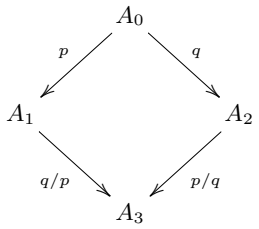


**Figure 3.** Residual Square

- *BACK THIS UP!*

In an ideal world, we would expect the residual function to have type $D\ a \to D\ a \to Maybe\ (D\ a)$, where the partiality comes from receiving two non-aligned patches.

But what if Bob and Alice changes the same cell in their CSV file? Then it is obvious that someone (human) have to chose which value to use in the final, merged, version.

Here we see that this residual operation is where conflicts are introduced in our theory.

### 3.1 Incorporating Conflicts

In order to track down these conflicts we need a more expressive patch data structure. We chose to parametrize $D$ over an abstract type and add another dummy constructor to it, much like we would do in a free monad construction. The actual data structure we use is presented in figure 4.

- *Show where conflicts arise and the two types of conflicts we identify.*

Note that the first constructor of $D$ just asks for a suitably indexed $A$. With this in mind, we can now start to define our residual operation.

- *Define* $Patch = D\ \bot$

```
_/_ : {n : ℕ}{t : Tel n}{ty : U n}
  → Patch t ty → Patch t ty → Maybe (D C t ty)
```

It is interesting to note that this residual operation is somewhat symmetric:

```
residual-symmetry-thm
  : {n : ℕ}{t : Tel n}{ty : U n}{k : D C t ty}
  → (d1 d2 : Patch t ty)
  → d1 / d2 ≡ just k
  → Σ (D C t ty → D C t ty)
      (λ op → d2 / d1 ≡ just (D-map C-sym (op k)))

residual-sym-stable : {n : ℕ}{t : Tel n}{ty : U n}{k : D C t ty}
  → (d1 d2 : Patch t ty)
  → d1 / d2 ≡ just k
  → forget <M> (d2 / d1) ≡ just (map (↓-map-↓ C-sym) (forget k))
```

This means that $p/q$ and $q/p$, although different, have the same conflicts (up to symmetry).

### 3.2 Solving Conflicts

- *This is highly dependent on the structure.*
  - *some structures might allow permutations, refactorings, etc... whereas others might not.*
- *How do we go generic? Free-monads to the rescue!*

```
mutual
  data D {a}(A : {n : ℕ} → Tel n → U n → Set a) : {n : ℕ} → Tel n → U n → Set a where
    D-A   : {n : ℕ}{t : Tel n}{ty : U n} → A t ty → D A t ty

    D-void : {n : ℕ}{t : Tel n} → D A t u1
    D-inl  : {n : ℕ}{t : Tel n}{a b : U n}
           → D A t a → D A t (a ⊕ b)
    D-inr  : {n : ℕ}{t : Tel n}{a b : U n}
           → D A t b → D A t (a ⊕ b)
    D-setl  : {n : ℕ}{t : Tel n}{a b : U n}
            → El U a t → El U b t → D A t (a ⊕ b)
    D-setr  : {n : ℕ}{t : Tel n}{a b : U n}
            → El U b t → El U a t → D A t (a ⊕ b)
    D-pair : {n : ℕ}{t : Tel n}{a b : U n}
           → D A t a → D A t b → D A t (a ⊗ b)
    D-mu : {n : ℕ}{t : Tel n}{a : U (suc n)}
      → List (Dμ A t a) → D A t (μ a)
    D-β : {n : ℕ}{t : Tel n}{F : U (suc n)}{x : U n}
      → D A (tcons x t) F → D A t (β F x)
    D-top : {n : ℕ}{t : Tel n}{a : U n}
      → D A t a → D A (tcons a t) vl
    D-pop : {n : ℕ}{t : Tel n}{a b : U n}
      → D A t b → D A (tcons a t) (wk b)

  data Dμ {a}(A : {n : ℕ} → Tel n → U n → Set a) : {n : ℕ} → Tel n → U (suc n) → Set a where
    Dμ-A  : {n : ℕ}{t : Tel n}{a : U (suc n)} → A t (μ a) → Dμ A t a
    Dμ-ins : {n : ℕ}{t : Tel n}{a : U (suc n)} → ValU a t → Dμ A t a
    Dμ-del : {n : ℕ}{t : Tel n}{a : U (suc n)} → ValU a t → Dμ A t a
    Dμ-cpy : {n : ℕ}{t : Tel n}{a : U (suc n)} → ValU a t → Dμ A t a
    Dμ-dwn : {n : ℕ}{t : Tel n}{a : U (suc n)} → ValU a t → D A t (β a u1) → Dμ A t a
```

**Figure 4.** Complete Definition of $D$

## 4. Sketching a Control Version System

- Different views over the same datatype will give different diffs.
- **newtype** annotations can provide a gread bunch of control over the algorithm.
- Directories are just rosetrees...

## 5. Related Work

- People have done similar things... or not.

## 6. Conclusion

- This is what we take out of it.