

Structure aware version control

A case study in Agda

Victor Cacciari Miraldo Wouter Swierstra

University of Utrecht

{v.cacciarimiraldo,w.s.swierstra} at uu.nl

Abstract

The widespread UNIX `diff3` program is a counterproductive tool for merging non-trivial edits of a given file. The problems with `diff3` arise from its unified view over files; all non-binary files are just lines of text. In the real world, however, we see that most data is much more than just lines of text, they have a great bunch of structure into it. This paper describes an approach, generic on the file structure, to obtain and merge changes in a much more elegant way. We use Agda to prototype and prove properties of our algorithms and Haskell to implement it. Moreover, we show how our approach scales, allowing programmers to easily define complex automatic conflict solvers.

Categories and Subject Descriptors D.1.1 [Programming Techniques]: Applicative (Functional) Programming; D.2.7 [Distribution, Maintenance, and Enhancement]: Version control; D.3.3 [Language Constructs and Features]: Data types and structures

General Terms Algorithms, Version Control, Agda, Haskell

Keywords Dependent types, Generic Programming, Edit distance, Patches

1. Introduction

Version control has become an indispensable tool in the development of modern software. There are various version control tools freely available, such as `git` or `mercurial`, that are used by thousands of developers worldwide. Collaborative repository hosting websites, such as GitHub and Bitbucket, haven't triggered a huge growth in open source development. [Add citations]

Yet all these tools are based on a simple, line-based diff algorithm to detect and merge changes made by individual developers. While such line-based diffs generally work well when monitoring small repositories, with small development teams, they tend to behave sub-optimally in many real life scenarios. A classical instance is indentation conflicts in source-code files. These conflicts happen precisely by comparing two files, line against line, will not reveal the real intent of an indentation.

Consider the following example CSV file, recording the marks and names three students:

Name	,	Number	,	Mark
Alice	,	440	,	7.0
Bob	,	593	,	6.5
Carroll	,	168	,	8.5

Adding a new line to this CSV file will not modify any existing entries and is unlikely to cause conflicts. Adding a new column storing the date of the exam, however, will change every line of the file and therefore will conflict with any other change to the file. Conceptually, however, this seems wrong: adding a column changes every line in the file, but leaves all the existing data unmodified. The only reason that this causes conflicts is that the *granularity of change* that version control tools use is not suitable for this kind of file.

This paper proposes a different approach to version control systems. Instead of relying on a single line-based diff algorithm, we will explore how to define a *generic* notion of change, together with algorithms for observing and combining such changes. To this end, this paper makes the following novel contributions:

- We define a universe representation for data and a *type-indexed* data type for representing edits to this structured data. The structure of the data closely resembles the way one defines datatypes in functional languages such as Haskell.
- We define generic algorithms for computing and applying a diff generically and prove they are correct with respect to each other.
- A notion of residual is used to propagate changes of different diffs, hence providing a bare mechanism for merging and conflict resolution.
- We illustrate how these ideas in Agda[?] have been implemented in a prototype Haskell tool, capable of automatically merging changes to structured data. This tool provides the user with the ability to define custom conflict resolution strategies when merging changes to structured data. For instance, we can automatically resolve refactoring conflicts.

Background

The generic diff problem is a very special case of the *edit distance* problem, which is concerned with computing the minimum cost of transforming an untyped tree A into an untyped tree B . Demaine provides a solution to the problem [?], improving the work of Klein [?]. This problem has been popularized in the particular case where the trees in question are, in fact, lists, in which case we call it the *least common subsequence* (LCS) problem [??]. The popular UNIX `diff` tool provides a solution to the LCS problem considering the edit operations to be inserting and deleting lines of text.

Our implementation follows a slightly different route, in which we choose not to worry too much about the *minimum* cost, but the one that better reflects the changes which are important to the use

case in question. In practice, we can see that the *diff* tool works great. Its problem is precisely conflict resolution, or more generally, its older brother *diff3* [?]. Conflict resolution is where our focus lies. We want to build a sound theory for patches that is highly customizable in its conflict solving capabilities, as it can be used for arbitrarily many file formats.

2. Structural Diffing

To make version control to be structure aware we need to define what structures we can handle. In our case, the structure is the universe of context free types, patching will be defined for its inhabitants. Our choice of universe has to do with how close to the universe of Haskell types we can get. This makes the translation from Agda to Haskell very simple, when we reach the point of implementing and running our algorithms, in section 4.

2.1 Context Free Datatypes

The universe of *context-free types*, in the sense of [?], is constructed following the grammar CF of context free types with a deBruijn representation for variables, figure 1.

$$\text{CF} ::= 1 \mid 0 \mid \text{CF} \times \text{CF} \mid \text{CF} + \text{CF} \mid \mu \text{CF} \mid \mathbb{N} \mid (\text{CF CF})$$

Figure 1. BNF for CF terms

In Agda, the CF universe is defined by:

```
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)
```

In order to make life easier we use a deBruijn indexing for our type variables. an element of $U\ n$ reads as a type with n free type variables. $u0$ and $u1$ represent the empty type and unit, respectively. Products and coproducts are represented by $_⊗_$ and $_⊕_$. Recursive types are created through μ . Type application is denoted by β . To control and select variables we use constructors that retrieve the *value* on top of the variable stack, vl , and that pop the variable stack, ignoring the top-most variable, wk . We decouple weakening wk from the variable occurrences vl and allow it anywhere in the code. This allows slightly more compact definitions later on.

Stating the language of our types is not enough. We need to specify its elements too, after all, they are the domain which we seek to define our algorithms for! Defining elements of fixed-point types make things a bit more complicated, check [?] for a more in-depth explanation of these details. The main idea, however, is that we need to take care of the free variables. For this need a *Telescope* of types in order to specify the elements of U and still pass the termination checker. Hence, we define the elements of U with respect to a *closing substitution*. Imagine we want to describe the elements of a type with n variables, $ty : U\ n$. We can only speak about this type once all n variables are bound to correspond to a given type. We need, then, t_1, t_2, \dots, t_n to pass as *arguments* to ty . Moreover, these types must have less free variables than ty itself, otherwise Agda can not check this substitution terminates. This list of types with decreasing type variables is defined through Tel :

```
data Tel : N → Set where
  tn1l : Tel 0
  tcons : {n : N} → U n → Tel n → Tel (suc n)
```

A value $(v : EU\ \{n\}\ ty\ t)$ reads roughly as: a value of type ty with n variables, applied to telescope t . At this point we can define the actual v 's that inhabit every code in $U\ n$. In Agda, the elements of U are defined by:

```
data EU : {n : N} → U n → Tel n → Set where
  unit : {n : N} {t : Tel n}
        → EU u1 t
  inl  : {n : N} {t : Tel n} {a b : U n}
        (x : EU a t) → EU (a ⊕ b) t
  inr  : {n : N} {t : Tel n} {a b : U n}
        (x : EU b t) → EU (a ⊕ b) t
  _'_  : {n : N} {t : Tel n} {a b : U n}
        → EU a t → EU b t → EU (a ⊗ b) t
  top  : {n : N} {t : Tel n} {a : U n}
        → EU a t → EU vl (tcons a t)
  pop  : {n : N} {t : Tel n} {a b : U n}
        → EU b t → EU (wk b) (tcons a t)
  mu   : {n : N} {t : Tel n} {a : U (suc n)}
        → EU a (tcons (μ a) t) → EU (μ a) t
  red  : {n : N} {t : Tel n} {F : U (suc n)} {x : U n}
        → EU F (tcons x t)
        → EU (β F x) t
```

The set EU of the elements of U is straight forward. We begin with some simple constructors to handle finite types: $unit$, inl , inr and $_'_$. Next, we define how to reference variables using pop and top . Finally, mu and red specify how to handle recursive types and type applications. We now have all the machinery we need to start defining types and their constructors inside Agda. Figure 2 shows the encoding of polymorphic lists and its constructors.

```
list : {n : N} → U (suc n)
list = μ (u1 ⊕ wk vl ⊗ vl)

CONS : {n : N} {t : Tel n} {a : U n}
      → EU a t → EU list (tcons a t)
      → EU list (tcons a t)
CONS x xs = mu (inr ((pop (top x)) , (top xs)))

NIL : {n : N} {t : Tel n} {a : U n}
      → EU list (tcons a t)
NIL = mu (inl unit)
```

Figure 2. The type of polymorphic lists and its constructors

Remember that our main objective is to speak about *how to track differences between elements of a given type*. So far we showed how to define the universe of context free types and the elements that inhabit it. With this definition, in Agda, we also get recursion principles for the elements of U and EU , which will let us define some generic operations over them. Finally, we define our diffing algorithms using only these (generic) operations, completely abstracting over types that can be defined in U and hence making them generic.

Some Generic Operations We can always view an element $EU\ ty\ t$ as a tree, this will be particularly usefull in section 2.3. The idea is that the telescope indicates how many levels our tree has, and which is the shape (type) of each subtree in each of those levels. Figure 3 illustrates this view for an element $EU\ ty\ (tcons\ t_1\ (tcons\ t_2\ tn1l))$. Note how we use vl to reference to the immediate children and wk to go one level deeper.

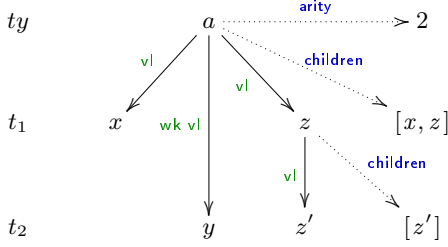


Figure 3. Children and Arity concepts illustrated

The intuition is that the children of an element is the list of immediate subtrees of that element, whereas its arity is how many such immediate subtrees are there. The actual Agda code of those operations is fairly complex, for it will be omitted. Their type is given by:

$$\begin{aligned} \text{children} &: \{n : \mathbb{N}\} \{t : \text{Tel } n\} \{a : \mathbf{U} (\text{succ } n)\} \{b : \mathbf{U} n\} \\ &\rightarrow \text{EIU } a (\text{tcons } b \, t) \rightarrow \text{List } (\text{EIU } b \, t) \end{aligned}$$

$$\begin{aligned} \text{arity} &: \{n : \mathbb{N}\} \{t : \text{Tel } n\} \{a : \mathbf{U} (\text{succ } n)\} \{b : \mathbf{U} n\} \\ &\rightarrow \text{EIU } a (\text{tcons } b \, t) \rightarrow \mathbb{N} \end{aligned}$$

A good advantage of Agda is that we can prove properties over our definitions:

$$\text{length } (\text{children } x) \equiv \text{arity } x$$

The unsuspecting reader might ask, then, why not *define* arity in this way? If we did define arity as `length · children` we would run into problems when writing types that *depend* on the arity of an element. Hence, we want `arity` to be defined directly by induction on its argument, making it structurally compatible with all other functions also defined by induction on `EIU`.

We can define many more operations over elements of `U`, but this is outside the scope of this paper. We refer the interested reader to the work of Altenkirch et al [?] for more. Our focus will now start shifting to our real objective: describing differences in the elements of `U`.

2.2 Patches over a Context Free Type

Let us consider a simple edit to a file containing students name, number and grade, as in figure 2.3. Assume that John, the responsible for the grades now knows that Carroll did drop out and that there was a mistake in Alice's grade. He then proceeds to edit the CSV file in order to reflect these changes.

Name	Number	Mark
Alice	440	7.0
Bob	593	6.5
Carroll	168	8.5

p_{John}

Name	Number	Mark
Alice	440	8.0
Bob	593	6.5

Figure 4. Sample Patch

Remember that a CSV structure is defined as a list of lists of cells. Let us assume that we have operations *enter*, *copy*, *change* and *del* that will: enter inside a structure; copy the structure con-

tents; change the contents or delete the structure, respectively. The patch p_{John} will then look like:

```
pJohn = [ enter [ copy, copy, copy, copy ]
, enter [ copy, copy, change 7.0 8.0, copy ]
, enter [ copy, copy, copy, copy ]
, del [ "Carroll", "168", "8.5" ]
, copy
]
```

Note how the patch closely follow the structure of the changes, which is exactly what we want! There is a single change, which happens at *Alice's mark* and a single deletion. Note also that we have to copy the end of both the inner and outer lists, this should not be confused with the list of *edit operations* describing the change.

Obviously, however, diffing CSV files is just the beginning. We shall now describe, formally, the actual *edit operations* that one can perform, by induction on the structure of `U`. The type of *diff's* is defined by `D`. It is indexed by a type and a telescope, which is the same as saying that we only define *diff's* for closed types¹. However, it also has a parameter `A`, this will be addressed later.

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

As we mentioned earlier, we are interested in analyzing the set of possible changes that can be made to objects of a type `T`. These changes depend on the structure of `T`, for the definition follows by induction on it.

If `T` is the Unit type, we can not modify it.

```
D-unit : {n : ℕ} {t : Tel n} → D A t u1
```

If `T` is a product, we need to provide *diffs* for both its components.

```
D-pair : {n : ℕ} {t : Tel n} {a b : U n}
→ D A t a → D A t b → D A t (a ⊗ b)
```

If `T` is a coproduct, things become slightly more interesting. There are four possible ways of modifying a coproduct, which are defined by:

```
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}
→ EIU a t → EIU b t → D A t (a ⊕ b)
D-setr : {n : ℕ} {t : Tel n} {a b : U n}
→ EIU b t → EIU a t → D A t (a ⊕ b)
```

Let us take a closer look at the *diffs* of a coproduct. There are four possibilities when modifying a coproduct $a \oplus b$. Given some *diff* p over a , we can always modify things that inhabit the left of the coproduct by `D-inl` p . Or we can change some given value *left* x into a *right* y , this is captured by `D-setl` $x \, y$. The case for `D-inr` and `D-setr` are analogous.

We also need some housekeeping definitions to make sure we handle all types defined by `U`.

```
D-β : {n : ℕ} {t : Tel n} {F : U (succ 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)
```

¹ Types that do not have any free type-variables

Fixed points are handled by a list of *edit operations*. We will discuss them in detail later on.

$D\text{-}\mu : \{n : \mathbb{N}\} \{t : \text{Tel } n\} \{a : U(\text{succ } n)\} \rightarrow \text{List } (D\mu A t a) \rightarrow D A t (\mu a)$

The aforementioned parameter A is used in a single constructor, allowing us to have a free-monad structure over D 's. This technique shows to be very useful for adding extra information, as we shall discuss, on section 3.1, for adding conflicts.

$D\text{-}A : \{n : \mathbb{N}\} \{t : \text{Tel } n\} \{ty : U n\} \rightarrow A t ty \rightarrow D A t ty$

If D is a free-monad it is also, in particular, a functor. For we have the equivalent mapping of a function on a D -structure, denoted by $D\text{-map}$. Moreover, we can also ignore the D -structure and fetch only the A 's inside, we call this *forget* : $\forall A t ty . D A t ty \rightarrow \text{List } A$. It is worth mentioning that all the indexes involved make the actual Agda type much more complicated. The intuition and the functionality stays the same, however. Finally, we define $\text{Patch } t ty$ as $D (\lambda _ _ \rightarrow \perp) t ty$. Meaning that a *Patch* is a D with *no* extra information.

2.3 Producing Patches

Given a generic definition of possible changes, the primary goal is to produce an instance of this possible changes, for two specific elements of a type T . We shall call this process *diffing*. It is important to note that our *gdiffl* function expects two elements of the same type! This contrasts with the work done by Vassena[?] and Lempink[?], where their *diff* takes objects of two different types.

For types which are not fixed points, the *gdiffl* functions follows the structure of the type:

$gdiffl : \{n : \mathbb{N}\} \{t : \text{Tel } n\} \{ty : U n\} \rightarrow \text{ElU } ty t \rightarrow \text{ElU } ty t \rightarrow \text{Patch } t ty$
 $gdiffl \{ty = v\} (top a) (top b) = D\text{-top } (gdiffl a b)$
 $gdiffl \{ty = wk u\} (pop a) (pop b) = D\text{-pop } (gdiffl a b)$
 $gdiffl \{ty = \beta F x\} (red a) (red b) = D\text{-}\beta (gdiffl a b)$
 $gdiffl \{ty = u1\} unit unit = D\text{-unit}$
 $gdiffl \{ty = ty \otimes tv\} (ay, av) (by, bv) = D\text{-pair } (gdiffl ay by) (gdiffl av bv)$
 $gdiffl \{ty = ty \oplus tv\} (inl ay) (inl by) = D\text{-inl } (gdiffl ay by)$
 $gdiffl \{ty = ty \oplus tv\} (inr av) (inr bv) = D\text{-inr } (gdiffl av bv)$
 $gdiffl \{ty = ty \oplus tv\} (inl ay) (inr bv) = D\text{-setl } ay bv$
 $gdiffl \{ty = ty \oplus tv\} (inr av) (inl by) = D\text{-setr } av by$
 $gdiffl \{ty = \mu ty\} a b = D\text{-}\mu (gdiffl (a :: []) (b :: []))$

Where the *gdiffl* takes care of handling fixed point values. The important remark here is that it operates over lists of elements, instead of single elements. This is due to the fact that the children of a fixed point element is a (possibly empty) list of fixed point elements.

Recursion Fixed-point types have a fundamental difference over the other type constructors in our universe. They can grow or shrink arbitrarily. We have to account for that when tracking differences between their elements. As we mentioned earlier, the *diff* of a fixed point is defined by a list of *edit operations*.

$data D\mu \{a\} (A : \{n : \mathbb{N}\} \rightarrow \text{Tel } n \rightarrow U n \rightarrow \text{Set } a) : \{n : \mathbb{N}\} \rightarrow \text{Tel } n \rightarrow U (\text{succ } n) \rightarrow \text{Set } a \text{ where}$

But the interesting bits are the *edit operations* we allow. We define $\text{Val } a t = \text{ElU } a (tcons u1 t)$ as the elements of type a where the recursive occurrences of μa are erased.

$D\mu\text{-ins} : \{n : \mathbb{N}\} \{t : \text{Tel } n\} \{a : U (\text{succ } n)\} \rightarrow \text{ValU } a t \rightarrow D\mu A t a$
 $D\mu\text{-del} : \{n : \mathbb{N}\} \{t : \text{Tel } n\} \{a : U (\text{succ } n)\} \rightarrow \text{ValU } a t \rightarrow D\mu A t a$
 $D\mu\text{-dwn} : \{n : \mathbb{N}\} \{t : \text{Tel } n\} \{a : U (\text{succ } n)\} \rightarrow D A (tcons u1 t) a \rightarrow D\mu A t a$

Again, we have a constructor for adding *extra* information, which is ignored in the case of *Patches*.

$D\mu\text{-}A : \{n : \mathbb{N}\} \{t : \text{Tel } n\} \{a : U (\text{succ } n)\} \rightarrow A t (\mu a) \rightarrow D\mu A t a$

The edit operations we allow are very simple. We can add or remove parts of a fixed-point or we can modify non-recursive parts of it. and instead of copying, we introduce a new constructor, $D\mu\text{-dwn}$, which is responsible for traversing down the type-structure. Copying is modeled by $D\mu\text{-dwn } (gdiffl x x)$. The intuition is that for every object x there is a *diff* that does not change x , we will look into this on section 2.3.

Before we delve into diffing fixed point values, we need some specialization of our generic operations for fixed points. Given that $\mu X.F X \approx F 1 \times \text{List } (\mu X.F X)$, that is, any inhabitant of a fixed-point type can be seen as a non-recursive head and a list of recursive children. We then make a specialized version of the *children* and *arity* functions, which lets us open and close fixed point values according to that *head and children* view.

$\text{Open}\mu : \{n : \mathbb{N}\} \rightarrow \text{Tel } n \rightarrow U (\text{succ } n) \rightarrow \text{Set}$
 $\text{Open}\mu t ty = \text{ElU } ty (tcons u1 t) \times \text{List } (\text{ElU } (\mu ty) t)$

$\mu\text{-open} : \{n : \mathbb{N}\} \{t : \text{Tel } n\} \{ty : U (\text{succ } n)\} \rightarrow \text{ElU } (\mu ty) t \rightarrow \text{Open}\mu t ty$

$\mu\text{-close} : \{n : \mathbb{N}\} \{t : \text{Tel } n\} \{ty : U (\text{succ } n)\} \rightarrow \text{Open}\mu t ty \rightarrow \text{Maybe } (\text{ElU } (\mu ty) t \times \text{List } (\text{ElU } (\mu ty) t))$

Although not explicit here, the list returned by $\mu\text{-open } x$ has length *arity* x . This is important since $\mu\text{-close}$ will consume exactly the *arity* x first elements of its input list. If the input list has less elements than *arity* x , we return *nothing*. A soundness lemma guarantees the correct behavior.

$\mu\text{-close-resp-arity} : \{n : \mathbb{N}\} \{t : \text{Tel } n\} \{ty : U (\text{succ } n)\} \{a : \text{ElU } (\mu ty) t\} \{hdA : \text{ElU } ty (tcons u1 t)\} \{chA l : \text{List } (\text{ElU } (\mu ty) t)\} \rightarrow \mu\text{-open } a \equiv (hdA, chA) \rightarrow \mu\text{-close } (hdA, chA ++ l) \equiv \text{just } (a, l)$

We denote the first component of an *opened* fixed point by its *value*, or *head*; whereas the second component by its children. These lemmas suggest that we handle fixed points in a serialized fashion. Since we never really know how many children will need to be handled in each step, we make *gdiffl* handle lists of elements or, forests, since every element is in fact a tree. Our algorithm, which was heavily inspired by [?], is then defined by:

```

gdiffL : {n : ℕ}{t : Tel n}{ty : U (suc n)}
→ List (EIO (μ ty) t) → List (EIO (μ ty) t) → Patch μ t ty
gdiffL [] [] = []
gdiffL [] (y :: ys) with μ-open y
... | hdY, chY = Dμ-ins hdY :: (gdiffL [] (chY ++ ys))
gdiffL (x :: xs) [] with μ-open x
... | hdX, chX = Dμ-del hdX :: (gdiffL (chX ++ xs) [])
gdiffL (x :: xs) (y :: ys)
= let
  hdX, chX = μ-open x
  hdY, chY = μ-open y
  d1 = Dμ-ins hdY :: (gdiffL (x :: xs) (chY ++ ys))
  d2 = Dμ-del hdX :: (gdiffL (chX ++ xs) (y :: ys))
  d3 = Dμ-dwn (gdiff hdX hdY) :: (gdiffL (chX ++ xs) (chY ++ ys))
in d1 ⊔μ d2 ⊔μ d3

```

The first three branches are simple. To transform $[]$ into $[]$, we do not need to perform any action; to transform $[]$ into $y : ys$, we need to insert the respective head and add the children to the *forest*; and to transform $x : xs$ into $[]$ we need to delete the respective values. The interesting case happens when we want to transform $x : xs$ into $y : ys$. Here we have three possible diffs that perform the required transformation. We want to choose the diff with the least *cost*, for we introduce an operator to do exactly that:

```

_ ⊔μ _ : {n : ℕ}{t : Tel n}{ty : U (suc n)}
→ Patch μ t ty → Patch μ t ty → Patch μ t ty
_ ⊔μ _ da db with costL da ≤? N costL db
... | yes _ = da
... | no _ = db

```

As we shall see in section 2.3.1, this notion of cost is very delicate. The idea, however, is simple. If the heads are equal we have $d3 = Dμ-dwn (gdiff hdX hdX) \dots$, which is how the copying instruction is encoded. It is clear that we want to copy as much as possible, so we will ensure that for all a , $gdiff a a$ has cost 0 whereas $Dμ-ins$ and $Dμ-del$ will have strictly positive cost. Therefore, we want to use the cost of a diff to drive the algorithm.

Note, however, that $gdiff a a$ is the patch that changes a into a . Well, but there are no changes to be made whatsoever. As it turns out, we do have some special patches that deserve some attention. They are the *identity patch* and the *inverse patch*. As we shall see later, the intuition from these two special patches will greatly influence how *cost* is defined.

The Identity Patch Given the definition of $gdiff$, it is not hard to see that whenever $x \equiv y$, we produce a patch without any $D-setl$, $D-setr$, $Dμ-ins$ or $Dμ-del$, as they are the only constructors of D that introduce *new information*. Hence we call these the *change-introduction* constructors. One can then spare the comparisons made by $gdiff$ and trivially define the identity patch for an object x , $gdiff-id x$, by induction on x . A good sanity check, however, is to prove that this intuition is in fact correct:

```

gdiff-id-correct
: {n : ℕ}{t : Tel n}{ty : U n}
→ (a : EIO ty t) → gdiff-id a ≡ gdiff a a

```

Later on we will talk about applying patches, for we will see that this is in fact the identity for x in *patch-space*.

The Inverse Patch If a patch $gdiff x y$ is not the identity, then it has *change-introduction* constructors. If we swap every $D-setl$ for $D-setr$, and $Dμ-ins$ for $Dμ-del$ and vice-versa, we get a patch that transforms y into x . We shall call this operation the inverse of a patch.

```

D-inv : {n : ℕ}{t : Tel n}{ty : U n}
→ Patch t ty → Patch t ty

```

As one would expect, $gdiff y x$ or $D-inv (gdiff x y)$ should be the same patch. In fact, we have that $gdiff y x \approx D-inv (gdiff x y)$. That is to say $gdiff y x$ is *observationally* the same as $D-inv (gdiff x y)$, but may not be identical. In the presence of equal cost alternatives they may make different choices.

Revisiting John's patch Recall the example given in figure . We can now write p_{John} with the actual result of diffing the CSV file before and after John makes his changes.

For readability purposes, we will omit the boilerplate *Patch* constructors. When diffing both versions of the CSV file, we get the patch that reflect John's changes over the initial file. Remember that $(Dμ-dwn (gdiff-id a))$ is merely copying a . The CSV structure is easily definable in U as $CSV = \beta \text{ list } (\beta \text{ list } X)$, for some appropriate atomic type X and p_{John} is then defined by:

```

pJohn = Dμ-dwn (gdiff-id "Name,...")
       Dμ-dwn ( Dμ-dwn (gdiff-id Alice)
               :: Dμ-dwn (gdiff-id 440)
               :: Dμ-dwn (gdiff 7.0 8.0)
               :: Dμ-dwn (gdiff-id[])
               [] )
       :: Dμ-dwn (gdiff-id "Bob,...")
       :: Dμ-del "Carroll,..."
       :: Dμ-dwn (gdiff-id[])
       []

```

2.3.1 The Cost Function

As we mentioned earlier, the cost function is one of the key pieces of the diff algorithm. It should assign a natural number to patches.

```

cost : {n : ℕ}{t : Tel n}{ty : U n} → Patch t ty → ℕ

```

The question is, how should we do this? The cost of transforming x and y intuitively leads one to think about *how far is x from y*. We see that the cost of a patch should not be too different from the notion of distance.

$$\text{dist } x y = \text{cost } (gdiff x y)$$

In order to achieve a meaningful definition, we will impose the specification that our *cost* function must make the distance we defined above into a metric. We then proceed to calculate the *cost* function from its specification. Remember that we call a function *dist* a *metric* iff:

$$\text{dist } x y = 0 \iff x = y \quad (1)$$

$$\text{dist } x y = \text{dist } y x \quad (2)$$

$$\text{dist } x y + \text{dist } y z \geq \text{dist } x z \quad (3)$$

Equation (1) tells that the cost of not changing anything must be 0, therefore the cost of every non-*change-introduction* constructor should be 0. The identity patch then has cost 0 by construction, as we seen it is exactly the patch with no *change-introduction* constructor.

Equation (2), on the other hand, tells that it should not matter whether we go from x to y or from y to x , the effort is the same. In patch-space, this means that the inverse of a patch should preserve its cost. Well, the inverse operation leaves everything unchanged but flips the *change-introduction* constructors to their dual counterpart. We will hence assign a cost $c_{\oplus} = \text{cost } D-setl =$

cost D-setr and $c_\mu = \text{cost D}\mu\text{-ins} = \text{cost D}\mu\text{-del}$ and guarantee this by construction already. Some care must be taken however, as if we define c_μ and c_\oplus as constants we will say that inserting a tiny object has the same cost of inserting a gigantic object! That is not what we are looking for in a fine-tuned diff algorithm. Let us then define $c_\oplus x y = \text{cost (D-setr } x y) = \text{cost (D-setl } x y)$ and $c_\mu x = \text{cost (D}\mu\text{-ins } x) = \text{cost (D}\mu\text{-del } x)$ so we can take this fine-tuning into account.

Equation (3) is concerned with composition of patches. The aggregate cost of changing x to y , and then y to z should be greater than or equal to changing x directly to z . We do not have a composition operation over patches yet, but we can see that this is already trivially satisfied. Let us denote the number of *change-introduction* constructors in a patch p by $\#p$. In the best case scenario, $\#(\text{gdiffl } x y) + \#(\text{gdiffl } y z) = \#(\text{gdiffl } x z)$, this is the situation in which the changes of x to y and from y to z are non-overlapping. If they are overlapping, then some changes made from x to y must be changed again from y to z , yielding $\#(\text{gdiffl } x y) + \#(\text{gdiffl } y z) > \#(\text{gdiffl } x z)$.

From equations (1) and (2) and from our definition of the identity patch and the inverse of a patch we already have that:

$$\begin{aligned} \text{gdiffl-id-cost} &: \{n : \mathbb{N}\} \{t : \text{Tel } n\} \{ty : \text{U } n\} \\ &\rightarrow (a : \text{EIU } ty \ t) \rightarrow \text{cost (gdiffl-id } a) \equiv 0 \end{aligned}$$

$$\begin{aligned} \text{D-inv-cost} &: \{n : \mathbb{N}\} \{t : \text{Tel } n\} \{ty : \text{U } n\} \\ &\rightarrow (d : \text{Patch } t \ ty) \\ &\rightarrow \text{cost } d \equiv \text{cost (D-inv } d) \end{aligned}$$

In order to finalize our definition, we just need to find an actual value for c_\oplus and c_μ . We have a lot of freedom to choose these values, however, yet, they are a critical part of the diff algorithm, since they will drive it to prefer some changes over others.

We will now calculate a relation that c_μ and c_\oplus need to satisfy for the diff algorithm to weight changes in a top-down manner. That is, we want the changes made to the outermost structure to be *more expensive* than the changes made to the innermost parts.

Let us then take a look at where the difference between c_μ and c_\oplus comes into play, and calculate from there. Assume we have stopped execution of gdiffl at the $d_1 \sqcup_\mu d_2 \sqcup_\mu d_3$ expression. Here we have three patches, that perform the same changes in different ways, and we have to choose one of them.

$$\begin{aligned} d_1 &= \text{D}\mu\text{-ins } hdY :: \text{gdiffl } (x :: xs) (chY \# ys) \\ d_2 &= \text{D}\mu\text{-del } hdX :: \text{gdiffl } (chX \# xs) (y :: ys) \\ d_3 &= \text{D}\mu\text{-dwn } (\text{gdiffl } hdX \ hdY) \\ &:: \text{gdiffl } (chX \# xs) (chY \# ys) \end{aligned}$$

We will only compare d_1 and d_3 , as the cost of inserting and deleting should be the same, the analysis for d_2 is analogous. By choosing d_1 , we would be opting to insert hdY instead of transforming hdX into hdY , this is preferable only when we do not have to delete hdX later on, in $\text{gdiffl } (x :: xs) (chY \# ys)$, as that would be a waste of information. Deleting hdX is inevitable when $hdX \notin chY \# ys$. Assuming without loss of generality that this deletion happens in the next step, we have:

$$\begin{aligned} d_1 &= \text{D}\mu\text{-ins } hdY :: \text{gdiffl } (x :: xs) (chY \# ys) \\ &= \text{D}\mu\text{-ins } hdY :: \text{gdiffl } (hdX :: chX \# xs) (chY \# ys) \\ &= \text{D}\mu\text{-ins } hdY :: \text{D}\mu\text{-del } hdX \\ &:: \text{gdiffl } (chX \# xs) (chY \# ys) \\ &= \text{D}\mu\text{-ins } hdY :: \text{D}\mu\text{-del } hdX :: \text{tail } d_3 \end{aligned}$$

Hence, $\text{cost } d_1$ is $c_\mu \text{hdX} + c_\mu \text{hdY} + w$, for $w = \text{cost (tail } d_3)$. Obviously, hdX and hdY are values of the same

type. Namely $hdX, hdY : \text{EIU } ty \ (\text{tcons } u1 \ t)$. Since we want to apply this to Haskell datatypes by the end of the day, it is acceptable to assume that ty is a coproduct of constructors. Hence hdX and hdY are values of the same finitary coproduct, representing the constructors of the fixed-point datatype. If hdX and hdY comes from different constructors, then² $hdX = i_j x'$ and $hdY = i_k y'$ where $j \neq k$. The patch from hdX to hdY will therefore involve a $\text{D-setl } x' y'$ or a $\text{D-setr } y' x'$, hence the cost of d_3 becomes $c_\oplus x' y' + w$. Remember that we are still in the situation where it is wise to delete and insert instead of recursively changing. The reasoning behind it is simple: since things are coming from a different constructors the structure of the outermost type is definitely changing, we want to reflect that! This means we need to select d_1 instead of d_3 , that is, we need to attribute a cost to d_1 that is strictly lower than the cost of d_3 :

$$\Leftrightarrow \begin{aligned} c_\mu (i_j x') + c_\mu (i_k y') + w &< c_\oplus (i_j x') (i_k y') + w \\ c_\mu (i_j x') + c_\mu (i_k y') &< c_\oplus (i_j x') (i_k y') \end{aligned}$$

If hdX and hdY come from the same constructor, on the other hand, the story is slightly different. We now have $hdX = i_j x'$ and $hdY = i_j y'$, the cost of d_1 still is $c_\mu (i_j x') + c_\mu (i_k y') + w$ but the cost of d_3 is $\text{dist } x' y' + w$, since $\text{gdiffl } hdX \ hdY$ will be $\text{gdiffl } x' y'$ preceded by a sequence of D-inr and D-inl , for hdX and hdY they come from the same coproduct injection, and these have cost 0. This is the situation that selecting d_3 is the best option, therefore we need:

$$\Leftrightarrow \begin{aligned} \text{dist } x' y' + w &< c_\mu (i_j x') + c_\mu (i_k y') + w \\ \text{dist } x' y' &< c_\mu (i_j x') + c_\mu (i_k y') \end{aligned}$$

In order to enforce this behavior on our diffing algorithm, we need to assign values to c_μ and c_\oplus that respects:

$$\text{dist } x' y' < c_\mu (i_j x') + c_\mu (i_k y') < c_\oplus (i_j x') (i_k y')$$

Note that there are an infinite amount of definitions that would fit here. This is indeed a central topic of future work, as the cost function is what drives the diffing algorithm. So far we have calculated a relation between c_μ and c_\oplus that will make the diff algorithm match in a top-down manner. That is, changes to the outermost type are seen as the heaviest changes. Different domains may require different relations. For a first generic implementation, however, this relation does make sense. Now is the time for finally assigning values to c_μ and c_\oplus and the diffing algorithm is completed. We simply define the cost function in such a way that it has to satisfy the imposed constraints. Firstly we define the size of an element:

$$\begin{aligned} \text{sizeEIU} &: \{n : \mathbb{N}\} \{t : \text{Tel } n\} \{u : \text{U } n\} \rightarrow \text{EIU } u \ t \rightarrow \mathbb{N} \\ \text{sizeEIU unit} &= 1 \\ \text{sizeEIU (inl } el) &= 1 + \text{sizeEIU } el \\ \text{sizeEIU (inr } el) &= 1 + \text{sizeEIU } el \\ \text{sizeEIU (ela, elb)} &= \text{sizeEIU } ela + \text{sizeEIU } elb \\ \text{sizeEIU (top } el) &= \text{sizeEIU } el \\ \text{sizeEIU (pop } el) &= \text{sizeEIU } el \\ \text{sizeEIU (mu } el) &= \text{let (hdE, chE) = } \mu\text{-open (mu } el) \\ &\quad \text{in sizeEIU } hdE + \text{foldr } _ + _ \ 0 \ (\text{map sizeEIU } chE) \\ \text{sizeEIU (red } el) &= \text{sizeEIU } el \end{aligned}$$

Finally, with $\text{costL} = \text{sum} \cdot \text{map } \text{cost}\mu$, we finish our cost function.

² We use i_j to denote the j -th injection into a finitary coproduct.

```

cost (D-A ())
cost D-unit = 0
cost (D-inl d) = cost d
cost (D-inr d) = cost d
cost (D-setl xa xb) = 2 * (sizeEIU xa + sizeEIU xb)
cost (D-setr xa xb) = 2 * (sizeEIU xa + sizeEIU xb)
cost (D-pair da db) = cost da + cost db
cost (D-β d) = cost d
cost (D-top d) = cost d
cost (D-pop d) = cost d
cost (D-mu l) = costL l

```

```

costμ (Dμ-A ())
costμ (Dμ-ins x) = 1 + sizeEIU x
costμ (Dμ-del x) = 1 + sizeEIU x
costμ (Dμ-dwn x) = cost x

```

2.4 Applying Patches

At this stage we are able to: work generically on a suitable universe; describe how elements of this universe can change and compute those changes. In order to make our framework useful, though, we need to be able to apply the patches we compute. The application of patches is easy, thus we only show the implementation for coproducts and fixed points here. The rest is very straightforward.

Patch application is a partial operation, however. A patch over T is an object that describe possible changes that can be made to objects of type T . The high-level idea is that diffing two objects $t_1, t_2 : T$ will produce a patch over T , whereas applying a patch over T to an object will produce a *Maybe* T . It is interesting to note that application can not be made total. Let's consider $T = X + Y$, and now consider a patch $(Left\ x) \xrightarrow{p} (Left\ x')$. What should be the result of applying p to a $(Right\ y)$? It is undefined!

```

gapply : {n : ℕ} {t : Tel n} {ty : U n}
  → Patch t ty → EIU ty t → Maybe (EIU ty t)
gapply (D-inl diff) (inl el) = inl <$> gapply diff el
gapply (D-inr diff) (inr el) = inr <$> gapply diff el
gapply (D-setl x y) (inl el) with x ?=U el
... | yes _ = just (inr y)
... | no _ = nothing
gapply (D-setr x y) (inr el) with y ?=U el
... | yes _ = just (inl x)
... | no _ = nothing
gapply (D-setr _ _) (inl _) = nothing
gapply (D-setl _ _) (inr _) = nothing
gapply (D-inl diff) (inr el) = nothing
gapply (D-inr diff) (inl el) = nothing
gapply {ty = μ ty} (D-mu d) el = gapplyL d (el :: []) >=> safeHead
:
gapplyL : {n : ℕ} {t : Tel n} {ty : U (suc n)}
  → Patchμ t ty → List (EIU (μ ty) t) → Maybe (List (EIU (μ ty) t))
gapplyL [] [] = just []
gapplyL [] _ = nothing
gapplyL (Dμ-A () :: _)
gapplyL (Dμ-ins x :: d) l = gapplyL d l >=> glns x
gapplyL (Dμ-del x :: d) l = gDel x l >=> gapplyL d
gapplyL (Dμ-dwn dx :: d) [] = nothing
gapplyL (Dμ-dwn dx :: d) (y :: l) with μ-open y
... | hdY, chY with gapply dx hdY
... | nothing = nothing
... | just y' = gapplyL d (chY ++ l) >=> glns y'

```

Where $\langle \$ \rangle$ is the applicative-style application for the *Maybe* monad; \gg is the usual bind for the *Maybe* monad and `safeHead` is the partial head function with type $[a] \rightarrow \text{Maybe } a$. In `gapplyL`, we have a `glns` function, which will get a value and a list of children of a fixed point, will try to μ -close it and add the result to the head of the remaining list. On the other hand, `gDel` will μ -open the head of the received list and compare its value with the received value, if they are equal it returns the tail of the input list appended to its children, if they are not equal it returns `nothing`.

Instead of presenting an example, let's provide some intuition for our `gapply` function. Looking at the `D-setl` case, for instance. `gapply (D-setl x y)` is expecting to transform a `inl x` into a `inr y`. Upon receiving a `inl` value, we need to check whether or not its contents are equal to x . If they are, we are good to go. If not, we have to return `nothing` as we cannot possibly know what to do. If we look instead on the `D-inl diff` branch, we see that it only succeeds upon receiving a `inl x`, given that `gapply diff` succeeds in modifying x .

The important part of application, nevertheless, is that it must produce the expected result. A correctness result guarantees that. Its proof is too big to be shown here but it has type:

```

correctness : {n : ℕ} {t : Tel n} {ty : U n}
  → (a b : EIU ty t)
  → gapply (gdiff a b) a ≡ just b

```

Combining `correctness` and `gdiff-id a ≡ gdiff a a` lemma, by transitivity, we see that our identity patch is in fact the identity. The *observational* equality of a patch and its inverse is obtained by transitivity with `correctness` and the following lemma:

```

D-inv-sound
  : {n : ℕ} {t : Tel n} {ty : U n}
  → (a b : EIU ty t)
  → gapply (D-inv (gdiff a b)) b ≡ just a

```

We have given algorithms for computing and applying differences over elements of a generic datatype. Moreover, we proved our algorithms are correct with respect to each other. This functionality is necessary for constructing a version control system, but it is by no means sufficient!

3. Patch Propagation

In a nutshell, any version control system must accomplish two tasks: (A) we need to be able to produce and apply patches and (B) we need to be able to merge different, concurrent, changes made to the same object. We have taken care of task (A) in the previous sections, and even though current VCS tools already excel at obtaining patches, there is a big lack of tools excelling at (B), that is, merging patches. All the structural information we are using in task (A) is, in fact, providing a lot more to help us at task (B), as we shall discuss in this section.

The task of merging changes arise when we have multiple users changing the same file at the same time. Imagine Bob and Alice perform concurrent edits in an object A_0 , which are captured by patches p and q . The center of the repository needs to keep only one copy of that object, but upon receiving the changes of both Bob and Alice we have:

$$A_1 \xleftarrow{p} A_0 \xrightarrow{q} A_2$$

Our idea, inspired by [?], is to incorporate the changes expressed by p into a new patch, namely one that is aimed at being applied somewhere already changed by q , and vice-versa, in such a way that they converge. We call this the residual patch. The diagram in figure 5 illustrates the result of merging p and q through propagation.

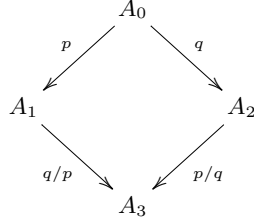


Figure 5. Residual patch square

The residual p/q of two patches p and q captures the notion of incorporating the changes made by p in an object that has already been modified by q .

In an ideal world, we would expect the residual function to have type $\text{Patch } t \text{ ty} \rightarrow \text{Patch } t \text{ ty} \rightarrow \text{Patch } t \text{ ty}$. Real life is more complicated. To begin with, it only makes sense to compute the residual of patches that are *aligned*, that is, they can be applied to the same input. For this, we make the residual function partial though the *Maybe* monad: $\text{Patch } t \text{ ty} \rightarrow \text{Patch } t \text{ ty} \rightarrow \text{Maybe } (\text{Patch } t \text{ ty})$ and define two patches to be aligned if and only if their residual returns a *just*.

Partiality solves just a few problems, what if, for instance, 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.

For this illustration, we will consider the conflicts that can arise from propagating the changes Alice made over the changes already made by Bob, that is, $p_{\text{Alice}}/p_{\text{Bob}}$.

- If Alice changes a_1 to a_2 and Bob changed a_1 to a_3 , with $a_2 \neq a_3$, we have an *update-update* conflict;
- If Alice deletes information that was changed by Bob we have an *delete-update* conflict;
- Last but not least, if Alice changes information that was deleted by Bob we have an *update-delete* conflict.
- If Alice adds information to a fixed-point, this is a *grow-left* conflict;
- When Bob added information to a fixed-point, which Alice didn't, a *grow-right* conflict arises;
- If both Alice and Bob add different information to a fixed-point, a *grow-left-right* conflict arises;

Most of the readers might be familiar with the *update-update*, *delete-update* and *update-delete* conflicts, as these are the most straight forward to be recognized as conflicts. We refer to these conflicts as *update* conflicts.

The *grow* conflicts are slightly more subtle. This class of conflicts corresponds to the *alignment table* that *diff3* calculates [?] before deciding which changes go where. The idea is that if Bob adds new information to a file, it is impossible that Alice changed it in any way, as it was not in the file when Alice was editing it, we then flag it as a conflict. The *grow-left* and *grow-right* are easy to handle, if the context allows, we could simply transform them into actual insertions or copies. They represent insertions made by Bob and Alice in *disjoint* places of the structure. A *grow-left-right* is more complex, as it corresponds to a overlap and we can not know for sure which should come first unless more information is provided. From the structure in our patch-space, we can already separate conflicts by the types they can occur on. An *update-update* conflict has to happen on a coproduct type, for it is the only type

which *Patches* over it can have multiple different options, whereas the rest are restricted to fixed-point types. In Agda,

```
data C : {n : N} → Tel n → U n → Set where
  UpdUpd : {n : N} {t : Tel n} {a b : U n}
    → EU (a ⊕ b) t → EU (a ⊕ b) t → EU (a ⊕ b) t
    → C t (a ⊕ b)
  DelUpd : {n : N} {t : Tel n} {a : U (suc n)}
    → ValU a t → ValU a t → C t (μ a)
  UpdDel : {n : N} {t : Tel n} {a : U (suc n)}
    → ValU a t → ValU a t → C t (μ a)
  GrowL : {n : N} {t : Tel n} {a : U (suc n)}
    → ValU a t → C t (μ a)
  GrowLR : {n : N} {t : Tel n} {a : U (suc n)}
    → ValU a t → ValU a t → C t (μ a)
  GrowR : {n : N} {t : Tel n} {a : U (suc n)}
    → ValU a t → C t (μ a)
```

3.1 Incorporating Conflicts

In order to track down these conflicts we need a more expressive patch data structure. We exploit D 's parameter for that matter. This approach has the advantage of separating conflicting from conflict-free patches on the type level, guaranteeing that we can only *apply* conflict-free patches.

The final type of our residual³ operation is:

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

We reiterate that the partiality comes from the fact the residual is not defined for non-aligned patches. The whole function is too big to be shown here, but explaining one of its cases can provide valuable intuition.

```
res (Dμ-dwn dx :: dp) (Dμ-del y :: dq)
  with gapply dx y
...| nothing = nothing
...| just y' with y ?= U y'
...| yes _ = res dp dq
...| no _ = _::_ (Dμ-A (UpdDel y' y)) <M> res dp dq
```

Here we are computing the residual:

$$(P_x = D\mu\text{-dwn } dx :: dp) / (P_y = D\mu\text{-del } y :: dq)$$

We want to describe how to apply the P_x changes to an object that has been modified by the P_y patch. Note the order is important! The first thing we do is to check whether or not the patch dx can be applied to y . If we can not apply dx to y , then patches P_x and P_y are non-aligned, we then simply return *nothing*. If we can apply dx to y , however, this will result in an object y' . We then need to compare y to y' , as if they are different we are in a *UpdDel* conflict situation. If they are equal, then dx is just *diff-id* y , that is, no changes were performed. To extend this to be applied to the object were y was deleted we simply suppress the *Dμ-del* and continue recursively. The remaining cases follow a similar reasoning process.

The attentive reader might have noticed a symmetric structure on conflicts. This is not at all by chance. In fact, we can prove that the residual of p/q have the same (modulo symmetry) conflicts as q/p . This proof goes in two steps. Firstly, *residual-symmetry* proves that if p and q are aligned, that is, $p/q \equiv \text{just } k$ for some k , then there exists a function op such that $q/p \equiv$

³ Our residual operation does not form a residual as in the Term Rewriting System sense[?]. It might, however, satisfy interesting properties. This is left as future work for now

`just (D-map C-sym (op k))`. We then prove, in `residual-sym-stable` that this function `op` does not introduce any new conflicts, it is purely structural. This could be made into a single result by proving that the type of `op` actually is $\forall A . D A t ty \rightarrow D A t ty$, we chose to split it for improved readability.

```
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 <$> (d2 / d1) ≡ just (map (↓-map-↓ C-sym) (forget k))
```

Here `↓-map-↓` takes care of hiding type indexes and `forget` is the canonical function with type $D A t ty \rightarrow List (\downarrow A)$, `↓_` encapsulates the type indexes of the different `A`'s we might come across.

Now, we can compute both p/q and q/p at the same time. It also backs up the intuition that using residuals or patch commutation (as in Darcs) is not significantly different. This means that p/q and q/p , although different, have the same conflicts (up to conflict symmetry).

Merge Strategies By looking at the type of *residual* we see that figure 5 does not really reflect what really happens with residuals. A more accurate picture is given in figure 6, where `op` is the function obtained by `residual-symmetry` and `e` is a special patch, lets call it a *merge strategy* for now.

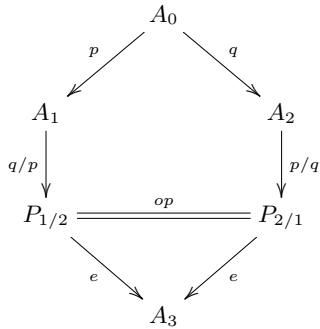


Figure 6. Residual patch square

Note that $P_{1/2}$ and $P_{2/1}$ are not really objects, as we can not apply a patch with conflicts. They are patches with conflicts. In order to more clearly discuss what is going on let us take a closer look at the types of the left path from A_0 to A_3 . We assume that $p, q : Patch A$ and $hip : q/p \equiv just k$ for some k , for the rest of this section.

In order to merge things, that is, to compute patches p' and q' that can be applied to A_1 and A_2 and produce the same A_3 we need to figure out what the aforementioned *merge strategy* actually is. Playing around with the types of our already defined functions we have:

$$\begin{array}{ccc} A & \xrightarrow{\text{flip } gdiff \ A_1} & Patch \ A \\ & \xrightarrow{(q/)} & Maybe \ (PatchC \ A) \\ & \xrightarrow{\text{fromJust } hip} & PatchC \ A \\ & \xrightarrow{e} & B \ (Patch \ A) \end{array}$$

By assumption and the types above, we see that a suitable type for the *merge strategy* e would be $PatchC \ A \rightarrow B \ (Patch \ A)$ for some behavior monad B . An interactive merge strategy would have $B = IO$, a partial merge strategy would have $B = Maybe$, etc. We can see that the design space is huge in order to define how to merge patches. Ideally we would like to have a library of *mergers* and a calculus for them, such that we can prove lemmas about the behavior of some *merge strategies*, that is, a bunch of *mergers* combined using different operators.

A simple pointwise *merge strategy* can be defined for a *merger* $m : \forall \{t \ ty\} \rightarrow C \ t \ ty \rightarrow D \perp_p \ t \ ty$, which can now be mapped over $D \ C \ t \ ty$ pointwise on its conflicts. We end up with an object of type $D \ (D \ \perp_p) \ t \ ty$. This is not a problem, however, since the free-monad structure on D provides us with a multiplication $\mu_D : D \ (D \ A) \ t \ ty \rightarrow D \ A \ t \ ty$. Therefore,

$$merge_{pw} \ m : D \ C \ t \ ty \xrightarrow{\mu_D \cdot D\text{-map } m} Patch \ t \ ty$$

would be one possible *merge strategy* using the *merger* m for removing the conflicts of a patch. Mapping a *merger* over the conflicting patch is by far not the only possible way of walking the tree, as we shall see in section 4. This opens up a lot of very interesting questions and paves the road to defining conflict resolution combinators. Allowing for a great degree of genericity in the base framework.

4. The Haskell Prototype

In sections 2.1 and 3 we have layered the foundations for creating a generic, structure aware, version control system. We proceed by illustrating these ideas with a prototype in Haskell, with an emphasis on its extended capability of handling non-trivial conflicts. A great advantage of using CF as a universe is that we are able to do generic-programming via typeclasses in Haskell.

The user has access to a typeclass *Diffable* a , which gives the basic diffing and merging functionality for objects of type a :

```
class (Sized a) => Diffable a where
  diff  :: a -> a -> Patch a
  apply :: Patch a -> a -> Maybe a
  res   :: Patch a -> Patch a -> Maybe (PatchC a)
  cost  :: Patch a -> Int
```

Where *Sized* a is a class providing the `sizeElU` function, presented in section 2.3.1; *Patch* is a GADT[?] reflecting our *Patch* type in Agda. We then proceed to provide instances by induction on the structure of a . Products and coproducts are trivial and follow immediately from the Agda code.

```
instance (Diffable a, Diffable b)
=> Diffable (a, b)
instance (Eq a, Eq b, Diffable a, Diffable b)
=> Diffable (Either a b)
```

Fixed points are not complicated, too. It is important that they support the same plugging and unplugging functionality as in Agda, though. We have to use explicit recursion since current Haskell's instance search does not have explicit type applications yet.

```
newtype Fix a = Fix { unFix :: a (Fix a) }
```

```

class (Eq (a ())) ⇒ HasAlg (a :: * → *) where
  ar  :: Fix a → Int
  ar  = length ∘ ch
  ch  :: Fix a → [Fix a]
  hd  :: Fix a → a ( )
  close :: (a ( ), [Fix a])
    → Maybe (Fix a, [Fix a])

instance (HasAlg (a :: * → *), Diffable (a ( )))
  ⇒ Diffable (Fix a)

```

All the other types can also be seen as sums-of-products. We then define a class and some template Haskell functionality to generate instances of *SOP* *a*. The *overlappable* pragma makes sure that Haskell’s instance search will give preference to the other *Diffable* instances, whenever the term head matches a product, coproduct atom or fixed-point.

```

class HasSOP (a :: *) where
  type SOP a :: *
  go :: a → SOP a
  og :: SOP a → a

instance {-# OVERLAPPABLE #-}
  (HasSOP a, Diffable (SOP a))
  ⇒ Diffable a

```

As the tool is still a very young prototype, we chose to omit implementation details. For those who wish to see these details, the code is available online⁴. There is, however, one extension we need to be able to handle built-in types. We have two additional constructors to *Patch* to handle atomic types:

```

newtype Atom a = Atom { unAtom :: a }
instance (Eq a) ⇒ Diffable (Atom a)

```

An *Atom* *a* is isomorphic to *a*. The difference is that it serves as a flag to the diff algorithm, telling it to treat the *a*’s atomically. That is, either they are equal or different, no inspection of their structure is made. As a result, there are only two possible ways to change an *Atom* *a*. We can either copy it, or change one *x* :: *a* into a *y* :: *a*.

4.1 A more involved proof of concept

In order to show the full potential of our approach, we will develop a simple example showing how one can encode and run a *refactoring* conflict solver for arbitrary datatypes. We will first introduce some simple definitions and then explore how refactoring can happen there. We

Our case study will be centered on CSV files with integers on their cells. The canonical representation of this CSV format is *T*. Moreover, we will also assume that the specific domain in which these files are used allows for refactorings.

```

type T = List (List (Atom Int))

```

Our *List* type is then defined as follows:

```

newtype L a x = L { unL :: Either ( ) (a, x) }
type List a = Fix (L a)

```

Again, *List* *a* is isomorphic to *[a]*, but it uses explicit recursion⁵ and hence has a *HasAlg* and *HasSOP* instance. Both of them are trivial and hence omitted. We hence have that *T* is isomorphic to *[[Int]]*. We will denote $\uparrow :: [[Int]] \rightarrow T$ as one of the witnesses of such isomorphism.

⁴ [HASKELL REPOSITORY]

⁵ The use of explicit recursion is what forces us to define *L* as a newtype, so that we can partially apply it.

We are now ready to go into our case study. Imagine both Alice and Bob clone a repository containing a single CSV file *l0* = \uparrow *[[1, 2], [3]]*. Both Alice and Bob make their changes to *lA* and *lB* respectively.

```

lA =  $\uparrow$  [[2], [3, 1]]
lB =  $\uparrow$  [[12, 2], [3]]

```

Here we see that Alice moved the cell containing the number 1 and Bob changed 1 to 12. Lets denote these patches by *pA* and *pB* respectively. In a simplified notation, they are represented by:

```

pA = [Dwn [Del 1, Cpy 2, Cpy []]
      , Dwn [Cpy 3, Ins 1, Cpy []]
      , Cpy []]
pB = [Dwn [Set 1 12, Cpy 2, Cpy []]
      , Cpy [3]
      , Cpy []]

```

We will now proceed to merge these changes automatically, following the approach on section 3, we want to propagate Alice’s changes over Bob’s patch and vice-versa. There will obviously be conflicts on those residuals. Here we illustrate a different way of traversing a patch with conflicts besides the free-monad multiplicative, as mentioned in section 3.1. Computing *pA* / *pB* yields:

```

pA / pB = [Dwn [DelUpd 1 12, Cpy 2, Cpy []]
           , Dwn [Cpy 3, GrowL 1, Cpy []]
           , Cpy []]

```

As we expected, there are two conflicts there! A *DelUpd* 1 12 and a *GrowL* 1 conflict on *pA* / *pB*. Note that the *GrowL* matches the deleted object on *DelUpd*. This is the *anatomy* of a refactoring conflict! Someone updated something that was moved by someone else. Moreover, from *residual-symmetry*, we know that that the conflicts in *pB* / *pA* are exactly *UpdDel* 12 1 and *GrowR* 1. The grow also matches the deleted object.

By permeating Bob’s changes over Alice’s refactor we would expect the the resulting CSV to be *lR* = \uparrow *[[2], [3, 12]]*. The functorial structure of patches provides us with exactly what we need to do so. The idea is that we traverse the patch structure twice. First we make a list of the *DelUpd* and *UpdDel* conflicts, then we do a second pass, now focusing on the *grow* conflicts and trying to match them with what was deleted. If they match, we either copy or insert the *updated* version of the object.

Recall 3.1, where we explain that conflict solving is comprised of a *merge strategy* combining different *mergers*. Although still not formulated in Agda, our Haskell prototype library already provides different *merge strategies* and *mergers*. It is worth mentioning that the actual code is slightly more complicated⁶, as the generic nature of the functions require some boilerplate code to typecheck but the main idea is precisely the same, for we will present a simplified version.

In the context provided by the current example, we will use a *merge strategy* *solvePWithCtx* with a *merger* *sRefactor*.

```

sRefactor      :: Cf a → [∀ x · Cf x] → Maybe (Patch a)
solvePWithCtx :: (Cf a → [∀ x · Cf x] → Maybe (Patch a))
  → PatchC b → PatchC b

```

The *solvePWithCtx* *merge strategy* will perform the aforementioned two traversals. The first one records the conflicts whereas the second one applies *sRefactor* to the conflicts. The *sRefactor* *merger*, in turn, will receive the list of all conflicts (context) and will try to match the *growths* with the *deletions*. Note that we return

⁶ We define a *subtyping* relation as a GADT, named *a >: b*, which specifies *b* as a subtype of *a*. The actual Haskell code uses this proofs extensively in order to typecheck and cast conflicts instead of the rank 2 types shown here.

a *PatchC* from the *merge strategy*. This happens since the *merge strategy* is *partial*. It will leave the conflicts it cant solve untouched. Predicate *resolved :: PatchC a → Maybe (Patch a)* casts it back to a patch if no conflict is present. We stress that the maximum we can do is provide the user with *merge strategies* and *mergers*, but since different domains will have different conflicts, it is up to the user to program the best strategy for that particular case. We leave as future work the development of an actual calculus of *mergers*, allowing one to actually prove their strategy will behave the way one expects.

We can now compute the patches *pAR* and *pBR*, to be applied to Alice’s and Bob’s copy in order to obtain the result, by:

```
Just pAR = resolved (solvePWithCtx sRefactor (pB / pA))
Just pBR = resolved (solvePWithCtx sRefactor (pA / pB))
```

Which evaluates to:

```
pAR = [ Cpy [2]
      ,   Dwn [Cpy 3, Set 1 12, Cpy []]
      ,   Cpy [] ]
pBR = [ Dwn [Del 12, Cpy 2, Cpy []]
      ,   Dwn [Cpy 3, Ins 12, Cpy []]
      ,   Cpy [] ]
```

And finally we can apply *pAR* to Alice’s copy and *pBR* to Bob’s copy and both will end up with the desired *lR* = ↑ [[2], [3, 12]] as a result.

As we can see from this not so simple example, our framework allows for a definition of a plethora of different conflict solving strategies. This fits very nicely with the *generic* part of the diff problem we propose to solve. In the future we would like to have a formal calculus of combinators for conflict solving, allowing different users to fully customize how their merge tool behaves.

5. Summary, Remarks and Related Work

On this paper we presented our approach to solving the generic diffing problem. We provided the theoretical foundations and created a Haskell prototype applying the proposed concepts. The diffing API can be made ready for all Haskell types, out of the box, with some simple Template Haskell, as all we need is the derivation of two trivial instances. We have also shown how this approach allows one to fully specialize conflict resolution for the domain in question. The work of Löh[?] and Vassena[?] are the most similar to our. We use a drastically different definition of patches, in order to have room for experimenting with conflict resolution.

Below we give a short comparison with other related work.

Antidiagonal Although easy to be confused with the diff problem, the antidiagonal is fundamentally different from the diff/apply specification. In [?], the antidiagonal for a type *T* is defined as a type *X* such that there exists $X \rightarrow T^2$. That is, *X* produces two *distinct* *T*’s, whereas a diff produces a *T* given another *T*.

Pijul The VCS Pijul is inspired by [?], where they use the free co-completion of a category to be able to treat merges as pushouts. In a categorical setting, the residual square (figure 5) looks like a pushout. The free co-completion is used to make sure that for every objects $A_i, i \in \{0, 1, 2\}$ the pushout exists. Still, the base category from which they build their results still handles files as a list of lines, thus providing an approach that does not take the file structure into account.

Darcs The canonical example of a *formal* VCS is Darcs [?]. The system itself is built around the *theory of patches* developed by the same team. A formalization of such theory using inverse semigroups can be found in [?]. They use auxiliary objects, called *Conflictors* to handle conflicting patches, however, it has

the same shortcoming for it handles files as lines of text and disregards their structure.

Finally, we address some issues and their respective solutions to the work done so far before concluding. The implementation of these solutions and the consequent evaluation of how they change our theory of patches is left as future work.

5.1 Cost, Inverses and Lattices

Back in section 2.3.1, where we calculated our cost function from a specification, we did not provide a formal proof that our definitions did in fact satisfy the relation we stated:

$$\text{dist } x' y' < c_\mu (i_j x') + c_\mu (i_k y') < c_\oplus (i_j x') (i_k y')$$

This is, in fact, deceptively complicated. Since \sqcup_μ is not commutative nor associative, in general, we do not have much room to reason about the result of a *gdiffl*. A more careful definition of \sqcup_μ that can provide more properties is paramount for a more careful study of the *cost* function. Defining \sqcup_μ in such a way that it gives us a lattice over patches and still respects patch inverses is very tricky, as for making it preserve inverses we need to have $\text{D-inv}(d_1 \sqcup_\mu d_2) \equiv \text{D-inv } d_1 \sqcup_\mu \text{D-inv } d_2$. A simpler option would be to aim for a quasi-metric *d* instead of a metric (dropping symmetry; equation (2)), this way inverses need not to distribute over \sqcup_μ and we can still define a metric *q*, but now with codomain \mathbb{Q} , as $q \ x \ y = \frac{1}{2} (d \ x \ y - d \ y \ x)$.

5.2 A remark on Type Safety

The main objectives of this project is to release a solid diffing and merging tool, that can provide formal guarantees, written in Haskell. The universe of user-defined Haskell types is smaller than context free types; in fact, we have fixed-points of sums-of-products. Therefore, we should be able to apply the knowledge acquired in Agda directly in Haskell. In fact, we did so! With a few adaptations here and there, to make the type-checker happy, the Haskell code is almost a direct translation. There is one minor detail we would like to point out in our approach so far. Our $\text{D}\mu$ type admits ill-formed patches. Consider the following simple example:

```
nat : U 0
nat = μ (u1 ⊕ v1)

ill-patch : Patch tnil nat
ill-patch = D-mu (D-mu-ins (inr (top void))) :: []

prf : ∀ el → gaply ill-patch el ≡ nothing
prf el = refl
```

On *ill-patch* we try to insert a *suc* constructor, which require one recursive argument, but then we simply end the patch. Agda realizes that this patch will lead nowhere in an instant.

Making $\text{D}\mu$ type-safe by construction should be simple. The type of the functor is always fixed, the telescope too. Hence they can become parameters. We just need to add two \mathbb{N} as indexes.

```
data Dμ {a} (A : {n : ℕ} → Tel n → U n → Set a)
  {n : ℕ} (t : Tel n) (ty : U (suc n))
  : ℕ → ℕ → Set where
```

Then, $\text{D}\mu \ A \ t \ ty \ i \ j$ will model a patch over elements of type $T = \text{E}(\mu \ ty) \ t$ and moreover, it expects a vector of length *i* of *T*’s and produces a vector of length *j* of *T*’s. This is very similar to how type-safety is guaranteed in [?], but since we have the types fixed, we just need the arity of their elements.

If we try to encode this in Agda, using the universe of context-free types, we run into a very subtle problem. In short, we can not

prove that if two elements of a recursive type come from the same constructor then they have the same arity. Mainly because this does not hold! This hinders $D\mu\text{-dwn}$ useless. Let us take a look at how one defines rose trees in CF:

$$\begin{aligned} \text{rt} &: \{n : \mathbb{N}\} \rightarrow \mathbf{U} (1 + n) \\ \text{rt} &= \mu (\text{wk } v_l \otimes \beta (\text{wk list}) v_l) \end{aligned}$$

Rose trees of a have a single constructor that takes an a and a list of rose trees of a to produce a single rose tree. Lets call its constructor RT . However, the arity of an element of a rose tree will vary. More precisely, it will be equal to the length of the list of recursive rose trees. We therefore can have two *heads* coming from the same constructor, as there is only one, with different arities, as we can see in:

$$\begin{aligned} r_1 &= RT \text{ unit } \text{NIL} \\ r_2 &= RT \text{ unit } (\text{CONS } r_1 \text{ NIL}) \end{aligned}$$

$$\begin{aligned} \text{problem} &: \text{arity } (\mu\text{-hd } r_1) \equiv 0 \times \text{arity } (\mu\text{-hd } r_2) \equiv 1 \\ \text{problem} &= \text{refl} , \text{refl} \end{aligned}$$

If we look at rt 's Haskell counterpart, `data RT a = RT a [RT a]`, we see that its arity should be zero, as the type $RT\ a$ does not appear immediately on the constructor, but only as an argument to the List type.

Although easy to describe, this problem is deeper than what meets the eye. On a separate example, consider a leaf tree, `ltree`, as defined below:

$$\begin{aligned} \text{ltree} &: \{n : \mathbb{N}\} \rightarrow \mathbf{U} (2 + n) \\ \text{ltree} &= \mu (\text{wk } (\text{wk } v_l) \oplus \text{wk } v_l \otimes v_l \otimes v_l) \end{aligned}$$

The Haskell equivalent, with explicit recursion, would be:

```
data LTreeF a b x = Leaf a | Fork b x x
type LTree a b    = Fix (LTreeF a b)
```

Now consider the following reduction.

$$\begin{aligned} \mathcal{M} &: \{n : \mathbb{N}\} \rightarrow \mathbf{U} (\text{suc } n) \\ \mathcal{M} &= \beta \text{ ltree list} \end{aligned}$$

In CF, due to the dependency introduced by the telescopes, this type of reduction is establishing a relation between the two type variables of `ltree`. Here we have the type of `ltrees` where the first type variable is actually a list of the second. In Haskell, this type would be defined as `type M a = LTree a [a]`.

To be able to encode this more delicate definition of arity we need to first divide our universe into sums-of-products, so that we have the notion of a constructor. Then change the definition of telescope to use closed types only (that is, types with `zero` type variables), not allowing this functional dependency between type variables. There are multiple ways to achieve this, we leave the exploration of these techniques as future work.

6. Conclusion

We believe that by incorporating the changes proposed in sections 5.1 and 5.2, we will be able to prove further results about our constructions. In particular we conjecture that our *residual* operation, section 3, constitutes, in fact, a residual system as in [? ?]. Moreover, we expect to be able to formulate more accurate properties about which conditions a *merge strategy*, section 3, must satisfy in order to converge. Moreover, we would like to have a formal categorical framework to speak about diffs.

From our proposals, we see that it is already possible to have much better merge tools to help automate the management of structured data. The applications are multiple. We can use our algorithms to create specialized merge tools for virtually every struc-

tured file format, as we just need a Haskell representation of this data to be able to diff it. This approach is easy to integrate on the already existing software version control systems but also allows us to develop one from scratch, for files and directories can also be represented in Haskell. Besides actual version control, we can also use the notion of `cost` we developed for a range of topics, given that we can always compute a non-trivial distance between values of an arbitrary datatype.