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

Modern version control systems are largely based on the UNIX `diff3` program for merging line-based edits on a given file. Unfortunately, this bias towards line-based edits does not work well for all file formats, leading to unnecessary conflicts. This paper describes a data type generic approach to version control that exploits a file's structure to create more precise diff and merge algorithms. We prototype and prove properties of these algorithms using the functional language in Agda; transcribing these definitions to Haskell yields a more scalable implementation.

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 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 source code in most programming languages, they tend observe unnecessary conflicts in many situations.

For exampl, consider the following example CSV file that records the marks, unique identification numbers, 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 the *granularity of change* that version control tools use is unsuitable for these files.

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 in Agda [**?**]. We have chosen a universe that closely resembles the algebraic data types that are definable in functional languages such as Haskell.
- We define generic algorithms for computing and applying a diff and prove that these algorithms satisfy several basic correctness properties.
- We define a notion of residual to propagate changes of different diffs on the same structure. This provides a basic mechanism for merging changes and resolving conflicts.
- We illustrate how our definitions in Agda may be used to implement 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.

•Add forward references in contributions to concrete subsections

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 a arbitrarily branching tree A into another, 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, when it is referred to 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 instead choose a cost model one that more accurately captures which

the changes are important to the specific data type in question. In practice, the *diff* tool works accurately manages to create patches by observing changes on a line-by-line basis. It is precisely when different changes must be merged, using tools such as *diff3* [**?**], that there is room for improvement.

2. Structural Diffing

To make version control systems aware of the *types* of data they manage, we need to collection of data types that may be versioned. More specifically, we will define a universe of context free types [**?**], whose values may be diffed and patched. Our choice of universe is intended to closely resemble the algebraic data types used by familiar functional languages. This will ease transition from Agda to a more scalable implementation in Haskell (Section 4).

2.1 Context Free Datatypes

The universe of *context-free types* [**?**], CF, is defined by the by the grammar in Figure 1.

$$\mathrm{CF} ::= 1 \mid 0 \mid \mathrm{CF} \times \mathrm{CF} \mid \mathrm{CF} + \mathrm{CF} \mid \mu x\ .\ \mathrm{CF} \mid x \mid (\mathrm{CF}\ \mathrm{CF})$$

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 will represent variables by De Bruijn indices; an element of `U n` reads as a type with n free type variables. The constructors `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 occurences `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. The main idea, however, is that we need to take define a suitable environment that captures the meaning of free variables. More specifically, we will use a `Tel`escope of types to specify the elements of `U`, while still satisfying Agda's termination checker. Hence, we define the elements of `U` with repsect 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, $t_1, t_2, \cdots, t_n$ to pass as *arguments* to ty. Moreover, these types must have less free variables then 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
  tnil  : Tel 0
  tcons : {n : ℕ} → U n → Tel n → Tel (suc n)
```

A value (v : `El U` $\{n\}$ 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 El U : {n : ℕ} → U n → Tel n → Set where
  unit : {n : ℕ}{t : Tel n}
       → El U u1 t
  inl  : {n : ℕ}{t : Tel n}{a b : U n}
       (x : El U a t) → El U (a ⊕ b) t
  inr  : {n : ℕ}{t : Tel n}{a b : U n}
       (x : El U b t) → El U (a ⊕ b) t
  _,_  : {n : ℕ}{t : Tel n}{a b : U n}
       → El U a t → El U b t → El U (a ⊗ b) t
  top  : {n : ℕ}{t : Tel n}{a : U n}
       → El U a t → El U vl (tcons a t)
  pop  : {n : ℕ}{t : Tel n}{a b : U n}
       → El U b t → El U (wk b) (tcons a t)
  mu   : {n : ℕ}{t : Tel n}{a : U (suc n)}
       → El U a (tcons (μ a) t) → El U (μ a) t
  red  : {n : ℕ}{t : Tel n}{F : U (suc n)}{x : U n}
       → El U F (tcons x t)
       → El U (β F x) t
```

The set `El U` of the elements of `U` is straightforward. We begin with some simple constructors to handle simple types, such as the unit type (`unit`), coproducts (`inl` and `inr`), and products (`_,_`). 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. For example, Figure 2 shows how to define a representation of polymorphic lists in this universe, together with its two constructors.

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

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

NIL : {n : ℕ}{t : Tel n}{a : U n}
  → El U 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 define *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. We can now define *generic* functions that operate on any type representible in this universe by induction over the representation type, `U`. In the coming sections, we define our diff algorithm using a handful of (generic) operations that we will define next.

Some Generic Operations We can always view an element el : `El U` ty t as a tree. The idea is that the telescope indicates how many 'levels' a tree may have, and which is the shape (type) of each subtree in each of those levels. Figure 3 illustrates this view for an element el : `El U` ty (`tcons` t_1 (`tcons` t_2 `tnil`)). Here, ty gives

the shape of the root whereas t_1 and t_2 gives the shape of levels 1 and 2. Note how we use vl to reference to the immediate children and wk to go one level deeper. Function arity is counting how many $\mathsf{El}\mathsf{U}t_1(\mathsf{tcons}\ t_2\ t)$ occurs in el.

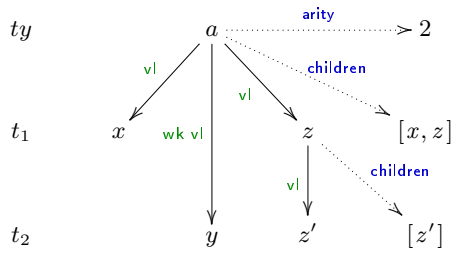

Figure 3. Children and Arity concepts illustrated

The intuition is that the children of an element is the list of immediate subtrees of that element, whereas its arity counts the number of immediate subtrees. The types of these two functions are given by:

```
children : {n : ℕ}{t : Tel n}{a : U (suc n)}{b : U n}
  → El U a (tcons b t) → List (El U b t)

arity : {n : ℕ}{t : Tel n}{a : U (suc n)}{b : U n}
  → El U a (tcons b t) → ℕ
```

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

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

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 El U.

With these auxiliary definitions in place, we can now turn our attention to the generic diff algorithm.

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. Suppose that Carroll drops out of the course and that there was a mistake in Alice's grade. We would like to edit the CSV file to reflect these changes.

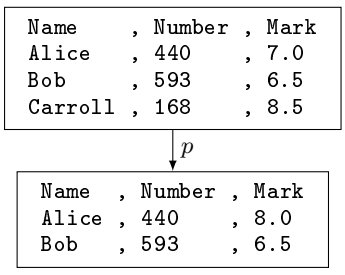

Figure 4. Sample Patch

Remember that a CSV structure is defined as a list of lists of cells. In what follows, we will define patches that operates on a specific CSV file. Such patches will be constructed from four primitive operations: *enter*, *copy*, *change* and *del*. The latter three should be familiar operations to copy a value, modify a value, or delete a value. The last operation, *enter*, will be used to inspect or edit the constituent parts of a composite data structure, such as the lines of a CSV file or the cells of a single line.

In our example, the patch p may be defined as follows:

```
p = [ 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 data. There is a single change, which happens in the third column of the second line and a single deletion. Note also that we have to copy the end of both the inner and outer lists – the last *copy* refers to the nil constructor terminating the list.

Obviously, however, diffing CSV files is just the beginning. We shall now formally describe the actual *edit operations* that one can perform by induction on the structure of U. The type of a *diff* is defined by the data type D. It is indexed by a type and a telescope. Finally, it also has a parameter A that we will come back to 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}
       → 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)
```

Let us take a closer look at the four potential changes that can be made to coproducts. There are four possibilities when modifying a coproduct $a \oplus b$. Given some diff p over a, we can always modify the left of the coproduct by D-inl p. Alternatively, 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 symmetrical.

Besides these basic types, we need a handful of constructors to handle variables:

```
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)
```

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

```
D-mu : {n : ℕ}{t : Tel n}{a : U (suc n)}
  → List (Dμ A t a) → D A t (μ a)
```

Finally, the aforementioned parameter A is used in a single constructor, ensuring our type for diffs forms a free monad. This constructor will be used for storing additional information about conflicts, as we shall see later (Section 3.1).

```
D-A : {n : ℕ}{t : Tel n}{ty : U n} → A t ty → 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-map. We traverse any D-structure and collect the values of type A it stores. We call this operation forget : $\forall A\ t\ ty\ .\ D\ A\ t\ ty \to$ List A. It is worth mentioning that all the indices involved make the actual Agda type a bit more complicated.

Finally, we define the type synonym Patch $t\ ty$ as D $(\lambda\ \_\ \_ \to \bot)\ t\ ty$. In other words, a Patch is a D structure that never uses the D-A constructor.

•fix formatting of D-A

2.3 Producing Patches

Next, we define a generic function gdiff that given two elements of a type in our universe, computes the patch recording their differences. It is important to note that our gdiff function expects two elements of the same type, in contrast to the work done by Vassena[**?**] and Lempsink[**?**].

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

```
gdiff : {n : ℕ}{t : Tel n}{ty : U n}
  → El U ty t → El U 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} unit unit = D-unit
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 :: []))
```

The only interesting branch is that for fixed-points, that is handled by the gdiffL function that operates over lists of elements, corresponding to the direct children of a given node.

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

But the interesting bits are the *edit operations* we allow. We define Val $a\ t$ = El U a (tcons u1 t) as the elements of type a where the recursive occurrences of $\mu\ a$ are replaced by unit values.

```
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-dwn : {n : ℕ}{t : Tel n}{a : U (suc n)}
  → D A (tcons u1 t) a → Dμ A t a
```

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

```
Dμ-A : {n : ℕ}{t : Tel n}{a : U (suc n)}
     → A t (μ a) → Dμ 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 its non-recursive parts. Instead of of copying, we introduce a new constructor, Dμ-dwn, which is responsible for traversing down the type-structure. Copying is modeled by Dμ-dwn (gdiff $x\ x$). For every object x we can define a patch Dμ-dwn that does not change x. We will return to this point in Section 2.3.

Before we delve into diffing fixed point values, we need some specialization of our generic operations for fixed points. Given that $\mu X.F\ X \approx F\ 1 \times$ List $(\mu X.F\ X)$, we may view any value of a fixed-point 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, in accordance with this observation.

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

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

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

Although not explicit here, the list returned by μ-open x has length arity x. This is important since μ-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.

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

We will refer to the first component of an *opened* fixed point as its *value*, or *head*; whereas we refer to the second component as 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 (ElU (μ ty) t) → List (ElU (μ ty) t) → Patchμ 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)}
  → Patchμ t ty → Patchμ t ty → Patchμ t ty
_⊔μ_ da db with costL da ⩽?-ℕ costL db
...| yes _ = da
...| no  _ = db
```

This operator compares two patches, returning the one with the lowest *cost*. As we shall see in section 2.4, this notion of cost is very delicate. Before we try to calculate a suitable definition of the cost function, however, we will briefly introduce two special patches and revisit our example.

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. The following property shows that our definition meets its specification:

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

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 vice-versa), 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. We can prove a slightly weaker statement, gdiff y x ≈ D-inv (gdiff x y). That is to say gdiff y x is *observationally* the same as D-inv (gdiff x y), but the two patches may not be identical. In the presence of equal cost alternatives they may make different choices.

Revisiting our example Recall the example given in Figure . We can define the patch p as the result of diffing the CSV file before and after our 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 our 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 $CSV = \beta$ list (β list X), for some appropriate atomic type X and p is then defined by:

```
p_John =  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[])
          []
```

2.4 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.

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

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:

$$dist\ x\ y = 0 \quad \Leftrightarrow \quad x = y \tag{1}$$

$$dist\ x\ y \quad = \quad dist\ y\ x \tag{2}$$

$$dist\ x\ y + dist\ y\ z \quad \geqslant \quad dist\ x\ z \tag{3}$$

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 $c_\oplus$ = cost D-setl = cost D-setr and c_μ = cost Dμ-ins = cost Dμ-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 $c_\oplus\ x\ y$ = cost (D-setr x y) = cost (D-setl x y) and $c_\mu\ x$ = cost (Dμ-ins x) = cost (Dμ-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, $\#(\mathsf{gdiff}\ x\ y) + \#(\mathsf{gdiff}\ y\ z) = \#(\mathsf{gdiff}\ x\ z)$, 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 $\#(\mathsf{gdiff}\ x\ y) + \#(\mathsf{gdiff}\ y\ z) > \#(\mathsf{gdiff}\ x\ z)$.

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

$$\begin{aligned}&\mathsf{gdiff\text{-}id\text{-}cost} : \{n : \mathbb{N}\}\{t : \mathsf{Tel}\ n\}\{ty : \mathsf{U}\ n\}\\ &\quad \to (a : \mathsf{El}\mathsf{U}\ ty\ t) \to \mathsf{cost}\ (\mathsf{gdiff\text{-}id}\ a) \equiv 0\end{aligned}$$

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

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.

$$\begin{aligned}d_1 &= \mathsf{D}\mu\text{-}\mathsf{ins}\ hdY :: \mathsf{gdiffL}\ (x :: xs)\ (chY +\!\!+ ys)\\ d_2 &= \mathsf{D}\mu\text{-}\mathsf{del}\ hdX :: \mathsf{gdiffL}\ (chX +\!\!+ xs)\ (y :: ys)\\ d_3 &= \mathsf{D}\mu\text{-}\mathsf{dwn}\ (\mathsf{gdiff}\ hdX\ hdY)\\ &\quad :: \mathsf{gdiffL}\ (chX +\!\!+ xs)\ (chY +\!\!+ ys)\end{aligned}$$

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 $\mathsf{gdiffL}\ (x :: xs)\ (chY +\!\!+ ys)$, 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:

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

Hence, $\mathsf{cost}\ d_1$ is $c_\mu\ \ hdX + c_\mu\ \ hdY + w$, for $w = \mathsf{cost}\ (\mathsf{tail}\ d_3)$. Obviously, hdX and hdY are values of the same type. Namely $hdX, hdY : \mathsf{El}\mathsf{U}\ ty\ (\mathsf{tcons}\ \mathsf{u1}\ t)$. 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, then[1] $hdX = i_j\ x'$ and $hdY = i_k\ y'$ where $j \neq k$. The patch from hdX to hdY will therefore involve a $\mathsf{D\text{-}setl}\ x'\ y'$ or a $\mathsf{D\text{-}setr}\ y'\ x'$, 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:

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

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 $\mathsf{dist}\ x'\ y' + w$, since $\mathsf{gdiff}\ hdX\ hdY$ will be $\mathsf{gdiff}\ x'\ y'$ preceded by a sequence of $\mathsf{D\text{-}inr}$ and $\mathsf{D\text{-}inr}$, 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:

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

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

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

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:

```
sizeElU : {n : ℕ}{t : Tel n}{u : U n} → ElU u t → ℕ
sizeElU unit       = 1
sizeElU (inl el)   = 1 + sizeElU el
sizeElU (inr el)   = 1 + sizeElU el
sizeElU (ela , elb) = sizeElU ela + sizeElU elb
sizeElU (top el)   = sizeElU el
sizeElU (pop el)   = sizeElU el
sizeElU (mu el)
  = let (hdE , chE) = μ-open (mu el)
    in sizeElU hdE + foldr _+_ 0 (map sizeElU chE)
sizeElU (red el)   = sizeElU el
```

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

[1] 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 * (sizeElU xa + sizeElU xb)
cost (D-setr xa xb) = 2 * (sizeElU xa + sizeElU 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 + sizeElU x
costμ (Dμ-del x) = 1 + sizeElU x
costμ (Dμ-dwn x) = cost x
```

2.5 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 → ElU ty t → Maybe (ElU 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 y x) (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 (ElU (μ ty) t) → Maybe (List (ElU (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 <$> is the applicative-style application for the *Maybe* monad; >>= is the usual bind for the *Maybe* monad and safeHead is the partial head function with type $[a] \to Maybe\ a$. 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, lets 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 : ElU ty t)
  → gapply (gdiff a b) a ≡ just b
```

Combining correctness and gdiff-id $a \equiv$ 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 : ElU 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.

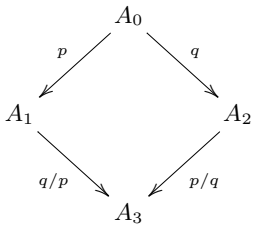

Figure 5. Residual patch square

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 `Patch t ty → Patch t ty → Patch t 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: `Patch t ty → Patch t ty → Maybe (Patch t 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_{Alice}/p_{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}
         → ElU (a ⊕ b) t → ElU (a ⊕ b) t → ElU (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 residual[2]. 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:

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

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, $p/q \equiv \mathsf{just}\ k$ for some k, then there exists a function op such that $q/p \equiv$

[2] 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 $ty \to$ 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 $\downarrow\!-map-\!\downarrow$ takes care of hiding type indexes and forget is the canonical function with type D A t $ty \to$ List ($\downarrow$ A), $\downarrow\_$ 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.

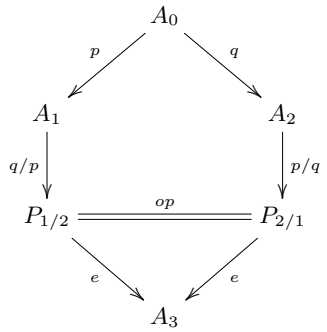

Figure 6. Residual patch square

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:

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

By assumption and the types above, we see that a suitable type for the *merge strategy* e would be PatchC $A \to$ 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* $m : \forall\{t\ ty\} \to$ C t $ty \to$ D $\bot_p$ 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 $\bot_p$) t ty. This is not a problem, however, since the free-monad structure on D provides us with a multiplication μ_D : D (D A) t $ty \to$ D A t ty. Therefore,

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

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 sizeE|U function, presented in section 2.4; *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 online[3]. 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 recursion[4] 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 $\uparrow :: [[Int]] \to T$ as one of the witnesses of such isomorphism.

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 = ↑ [[2], [3, 1]]
lB = ↑ [[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 multiplication, as mentioned in section 3.1. Computing $pA \mathbin{/} pB$ 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 $pA \mathbin{/} pB$. 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 $pB \mathbin{/} pA$ 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 complicated[5], 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

[3] [HASKELL REPOSITORY]

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

[5] 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 $resolved :: PatchC\ a \to Maybe\ (Patch\ a)$ 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:

$$\begin{aligned} pAR = [&\ Cpy\ [2] \\ ,&\ Dwn\ [Cpy\ 3, Set\ 1\ 12, Cpy\ []] \\ ,&\ Cpy\ []] \\ pBR = [&\ Dwn\ [Del\ 12, Cpy\ 2, Cpy\ []] \\ ,&\ Dwn\ [Cpy\ 3, Ins\ 12, Cpy\ []] \\ ,&\ Cpy\ []] \end{aligned}$$

And finally we can apply pAR to Alice's copy and pBR to Bob's copy and both will end up with the desired $lR = \uparrow [[2], [3, 12]]$ 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 $X \to T^2$. 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 2.4, 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:

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

This is, in fact, deceptively complicated. Since $\_\sqcup_\mu\_$ 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 $\_\sqcup_\mu\_$ that can provide more properties is paramount for a more careful study of the cost function. Defining $\_\sqcup_\mu\_$ 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 $\mathsf{D\text{-}inv}(d_1 \sqcup_\mu d_2) \equiv \mathsf{D\text{-}inv}\ d_1 \sqcup_\mu \mathsf{D\text{-}inv}\ d_2$. 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 $\_\sqcup_\mu\_$ 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 $\mathsf{D}\mu$ type admits ill-formed patches. Consider the following simple example:

```
nat : U 0
nat = μ (u1 ⊕ vl)

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

prf : ∀ el → gapply 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 $\mathsf{D}\mu$ 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, $\mathsf{D}\mu\ A\ t\ ty\ i\ j$ will model a patch over elements of type $T = \mathsf{El U}(\mu\ ty)\ t$ 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 Dμ-dwn useless. Let us take a look at how one defines rose trees in CF:

```
rt : {n : ℕ} → U (1 + n)
rt = μ (wk vl ⊗ β (wk list) vl)
```

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:

```
r₁ = RT unit NIL
r₂ = RT unit (CONS r₁ NIL)

problem : arity (μ-hd r₁) ≡ 0 × arity (μ-hd r₂) ≡ 1
problem = refl , refl
```

If we look at rt's Haskell counterpart, $\mathbf{data}\ RT\ a = RT\ a\ [RT\ a]$, 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:

```
ltree : {n : ℕ} → U (2 + n)
ltree = μ (wk (wk vl) ⊕ wk vl ⊗ vl ⊗ vl)
```

The Haskell equivalent, with explicit recursion, would be:

$\mathbf{data}\ LTreeF\ a\ b\ x = Leaf\ a \mid Fork\ b\ x\ x$
$\mathbf{type}\ LTree\ a\ b \quad = Fix\ (LTreeF\ a\ b)$

Now consider the following reduction.

```
𝓜 : {n : ℕ} → U (suc n)
𝓜 = β ltree list
```

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 $\mathbf{type}\ \mathcal{M}\ a = LTree\ a\ [a]$.

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 structured 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.