Structure-aware version control

A generic approach using Agda

Author 1 Author 2

Some Institution

{email1,email2} at some.institution

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 dependently typed language 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.

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 example, 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 [14]. We have chosen a universe that closely resembles the algebraic data types that are definable in functional languages such as Haskell (Section 2.1).
- We define generic algorithms for computing and applying a diff and prove that these algorithms satisfy several basic correctness properties (Section 2.3).
- 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 (Section 3).
- 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 (Section 4).

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 [7], improving the work of Klein [10]. 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 [4, 6]. 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* [9], that there is room for improvement.

2. Structural Diffing

To make version control systems aware of the *types* of data they manage, we need the collection of data types that may be versioned. More specifically, we will define a universe of context free types [2], 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* [2], CF, is defined by the by the grammar in Figure 1.

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

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
  def : {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 `def`. 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. 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 Telescope of types to specify the elements of `U`, while still satisfying Agda's 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, $t_1, t_2, \cdots, t_n$ 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 (def 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 representable 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 `El U`t_1(`tcons` t_2 t) occurs in el.

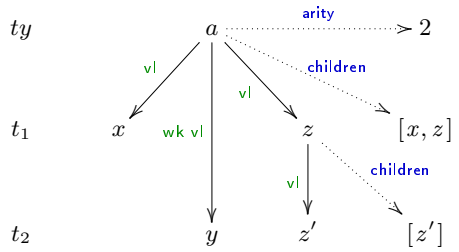

Figure 3. Children and Arity concepts illustrated

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.

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

p

```
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. 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 : {n : ℕ} → Tel n → U n → Set)
  : {n : ℕ} → Tel n → U n → Set 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 cannot 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 inl x into a inr 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-def : {n : ℕ}{t : Tel n}{F : U (suc n)}{x : U n}
  → D A (tcons x t) F → D A t (def 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*, List (Dμ A t a). 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, D-A, 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
```

Finally, we define the type synonym Patch t ty as D (λ _ _ → ⊥) t ty. In other words, a Patch is a D structure that never uses the D-A constructor.

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. 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}
  → ElU ty t → ElU ty t → Patch t ty
gdiff {ty = vl} (top a) (top b)       = D-top (gdiff a b)
gdiff {ty = wk u} (pop a) (pop b)     = D-pop (gdiff a b)
gdiff {ty = def F x} (red a) (red b) = D-def (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. Let us now see how to handle the diff of fixed-points.

Recursion Fixed-point types have a fundamental difference over the other type constructors in our universe. They can grow or shrink arbitrarily. Just like for values of coproducts, where we had multiple ways of changing them, we have three possible changes we can make to the value of a fixed-point. This time, however, they are not mutually exclusive.

For example, imagine we are making changes in an element of def list bool. One such change is depicted in Figure 5, where the list grows in the middle, by CONS ff (CONS tt ·) and shrinks in the end.

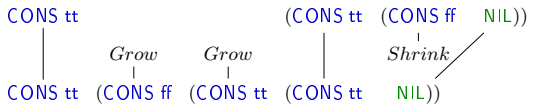

Figure 5. Growing and Shrinking a fixed-point

Note that Figure 5 is not the only possible representation of such change by means of grows and shrinks. In fact, the `diff3` tool pre-computes an alignment table for identifying where the file grows and shrinks before computing the actual differences. We chose to dynamically discover where the fixed-point value grows and shrinks instead of pre-computing such a table, since types other than list give rise to a grow-shrink pattern that do not resemble a table, but the structure of the respective type itself. Although we cannot represent these patterns in a uniform fashion for all types, we can fix the way in which we traverse a type for growing and shrinking it. Hence, we can model the diff of a fixed-point as a list of atomic *edit operations*:

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

And here we define the *edit operations* we allow. Whenever the value grows it means something was inserted, whenever the value shrinks, it means something was deleted. We define Val a t = ElU a (tcons u1 t) as the elements of type a where the recursive occurrences of μ 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, analogous to *enter* used in Figure 2.3. 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 $\mu X.F\ X \approx F\ 1 \times \text{List}\ (\mu X.F\ X)$, 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 = ElU ty (tcons u1 t) × List (ElU (μ ty) t)

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

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

Although 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 : ElU (μ ty) t}
    {hdA : ElU ty (tcons u1 t)}{chA l : List (ElU (μ ty) t)}
  → μ-open a ≡ (hdA , chA)
  → μ-close (hdA , chA ++ l) ≡ just (a , l)
```

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 do a pre-order traversal of the tree corresponding to the fixed-point value in question. 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 [11], is then defined by:

```
gdiffL : {n : ℕ}{t : Tel n}{ty : U (suc n)}
  → List (El U (μ ty) t) → List (El U (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
```

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 : El U 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 $CSV = \text{def list (def list } X)$, for some appropriate atomic type X and p is then defined by:

```
p  =  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 other words, inverting a patch should preserve its cost. 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. This guarantees the second property by construction. If we define c_μ and $c_\oplus$ as constants, however, the cost of inserting a small subtree will have the same cost as inserting a very large subtree. This is probably undesirable and may lead to unexpected behavior. Instead of constants, $c_\oplus$ and c_μ, we will instead try to define a pair of functions, $c_\oplus\ x\ y$ = cost (D-setr x y) = cost (D-setl x y) and $c_\mu\ x$ = cost (Dμ-ins x) = cost (Dμ-del x), that may take the size of the arguments 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:

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

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

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 and they are a critical part of the diff algorithm. The choice of cost model will prefer certain changes over others.

We will now calculate a relation that c_μ and $c_\oplus$ need to satisfy for the diff algorithm to weigh 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. For example, when considering the CSV file example, we consider inserting a new line to be a more expensive operation than updating a single cell.

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 when computing $\mathsf{gdiffL}\ (x :: xs)\ (chY +\!\!+ ys)$. Deleting hdX is inevitable when hdX does not occur as a subtree in the remaining structures to diff, that is, $hdX \notin chY +\!\!+ ys$. Assuming without loss of generality that this deletion happens in the next step, we can calculate:

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

Hence, $\mathsf{cost}\ d_1$ is $c_\mu\ \ hdX + c_\mu\ \ hdY + w$, for $w = \mathsf{cost}\ (\mathsf{tail}\ d_3)$. Here hdX and hdY are values of the same type, $\mathsf{EIU}\ ty\ (\mathsf{tcons}\ u1\ t)$. As our data types will be sums-of-products, we will assume that ty is a coproduct of constructors. Hence hdX and hdY are values of the same finitary coproduct, representing the constructors of a (recursive) data type. In what follows we will use i_j to denote the j-th injection into a finitary coproduct. If hdX and hdY comes from different constructors, then $hdX = i_j\ x'$ 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 $c_\oplus\ x'\ y' + w$. The reasoning behind this choice is simple: since the outermost constructor is changing, the cost of this change should reflect this. As a result, 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. Note that we are *not* defining the cost function yet, but calculating the relations it needs to satisfy in order to behave the way we want.

$$\begin{aligned}&&\mathsf{cost}\ d_1 &< \mathsf{cost}\ d_3\\ \Leftrightarrow\ & c_\mu\ (i_j\ x') + c_\mu\ (i_k\ y') + w &< c_\oplus\ (i_j\ x')\ (i_k\ y') + w\\ \Leftarrow\ & c_\mu\ (i_j\ x') + c_\mu\ (i_k\ y') &< c_\oplus\ (i_j\ x')\ (i_k\ y')\end{aligned}$$

If hdX and hdY come from the same constructor, on the other hand, the story is slightly different. In this scenario we prefer to choose d_3 over d_1, as we want to preserver the constructor information. 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 will be $\mathsf{cost}\ (\mathsf{gdiff}\ (i_j\ x')\ (i_j\ y')) + w$. Well, $\mathsf{gdiff}\ (i_j\ x')\ (i_j\ y')$ will reduce to $\mathsf{gdiff}\ x'\ y'$ preceded by a sequence of $\mathsf{D\text{-}inr}$ and $\mathsf{D\text{-}inr}$. Hence, $\mathsf{cost}\ d_3 = \mathsf{cost}\ (\mathsf{gdiff}\ x'\ y') + w$.

Remember that we want to select d_3 instead of d_1, based on their costs. The way to do so is to enforce that d_3 will have a strictly smaller cost than d_1. We hence calculate the relation our cost function will need to respect:

$$\begin{aligned}&&\mathsf{cost}\ d_3 &< \mathsf{cost}\ d_1\\ \Leftrightarrow\ & \mathsf{dist}\ x'\ y' + w &< c_\mu\ (i_j\ x') + c_\mu\ (i_j\ y') + w\\ \Leftarrow\ & \mathsf{dist}\ x'\ y' &< c_\mu\ (i_j\ x') + c_\mu\ (i_j\ y')\end{aligned}$$

Record that our objective was to calculate a relation that the cost function needs to satisfy in order to preserve as many constructors as possible. We did so by analyzing the case in which we want gdiff to preserve the constructor against the case where we want gdiff to delete or insert new constructors. Now we can finally assign values to c_μ and $c_\oplus$. By transitivity and the relations calculated above we get:

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

Note that there are many definitions that satisfy the specification we have outlined above. So far we have calculated a relation between c_μ and $c_\oplus$ that encourages the diff algorithm to favor (smaller) changes further down in the tree.

To complete our definition, we still need to choose c_μ and $c_\oplus$. We simply define the cost function in such a way that it has to satisfy the imposed constraints. To do so, we begin by defining the size of an element:

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

Finally, we define $\mathsf{costL} = \mathsf{sum} \cdot \mathsf{map}\ \mathsf{cost}\mu$. This completes the definition of the diff algorithm.

```
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-def 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
```

The choice of cost function determines how the diff algorithm works; finding further evidence that the choice we have made here works well in practice requires further work. Different domains may require different relations.

2.5 Applying Patches

Although we have now defined an algorithm to *compute* a patch, we have not yet defined a function to apply a patch to a given structure. For the most part, the generic function that does this is fairly straightforward. We will cover two typical cases, coproducts and fixed points, in some detail here.

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. Consider the case for coproducts, that is, $T = X + Y$. Suppose we have a patch p modifying one component of the coproduct, mapping (inl x) to (inl x'). What should be the result of applying p to the value (inr y)? As there is no sensible value that we can return, we instead choose to make the application of patches a partial function that returns a value of Maybe T.

The overall idea is that a Patch T specify a bunch of instructions to transform a given $t_1 : T$ into a $t_2 : T$. The gapply function is the machine that understands these instructions and performs them on t_1, yielding t_2. For example, consider the case for the D-setl constructor, which is expecting to transform an 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 this holds, we can simply return inr y as intended. If not, we fail and return nothing.

The definition of the gapply function proceeds by induction on the patch:

```
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
  ⋮
```

Where <$> is the applicative-style application for the *Maybe* monad; >>= is the usual bind for the *Maybe* monad and safeHead is the partial function of type $[a] \to Maybe\ a$ that returns the first element of a list, when it exists. Despite the numerous cases that must be handled, the definition of gapply for coproducts is reasonably straightforward.

```
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'
```

The case for fixed points is handled by the gapplyL function. This function uses two auxiliary functions, glns and gDel. They provide an easy way to μ-close and to μ-open a head and a list fixed-point values.

```
glns x l with μ-close (x , l)
... | nothing = nothing
... | just (r , l') = just (r :: l')


gDel x [] = nothing
gDel x (y :: ys) with x == (μ-hd y)
... | True  = just (μ-ch y ++ ys)
... | False = nothing
```

Intuitively, we can see glns creates a new fixed-point value. It uses the supplied head and the corresponding number of children from the supplied list. We then just return a list, with the newly created value and the remaining list. The gDel function is the dual case. It will delete the head of the head of the supplied list if it matches the supplied head.

Given this definition of gapply, we can now formally prove the following property:

```
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. For a structure aware version control tool, however, we will also need to reconcile different changes.

3. Patch Propagation

A version control system must handle three separate tasks: it must produce patches, based on the changes to a file; it must apply patches to a file; and, finally, it must merge patches made to the same file. In the previous section, we defined generic algorithms for

creating and applying patches. In this section, we turn our attention to the final point: merging different patches. It is precisely here that we expect to be able to exploit the structure of files to avoid unnecessary conflicts.

The task of merging changes arise when we have multiple users changing the same file at the same time. Imagine Bob and Alice perform edits on a file A_0, resulting in two patches p and q. We might visualize this situation in the following diagram:

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

Our idea, inspired by Tieleman [18], is to incorporate the changes made by p into a new patch, that may be applied to A_2 which we will call the residual of p after q, denoted by q/p. Similarly, we can compute the residual of p/q. The diagram in Figure 6 illustrates the result of merging the patches p and q using their respective residuals:

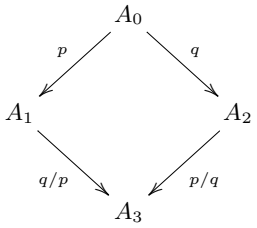

Figure 6. Residual patch square

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.

Ideally, we would hope the residual function to have type $\mathsf{Patch}\ t\ ty \to \mathsf{Patch}\ t\ ty \to \mathsf{Patch}\ t\ ty$. Unfortunately, we cannot define a total notion of residual as it only makes sense to compute the residual of patches that are *aligned*, that is, they can be applied to the same input. Hence, we make the residual function partial though the *Maybe* monad: $\mathsf{Patch}\ t\ ty \to \mathsf{Patch}\ t\ ty \to \mathsf{Maybe}\ (\mathsf{Patch}\ t\ ty)$ and define two patches to be aligned if and only if their residual returns a just. For example, for all x, y the patches $\mathsf{D\text{-}setl}\ x\ y$ and $\mathsf{D\text{-}setr}\ y\ x$ are *not* aligned. The former expects a $\mathsf{inl}\ x$ as input whereas the later expects a $\mathsf{inr}\ y$.

Even if we restrict ourselves to a partial residual function, there may be other issues that arise. In particular, suppose that both Bob and Alice change the same cell in the CSV file. There is no reason to favor one particular change over another. In that case, we report a *conflict* and leave it to the end user to choose which value to use final result.

We will now consider the different kinds of conflicts that can arise from propagating the changes Alice made over the changes already made by Bob, that is in computing the residual 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;
- If Alice changes information that was deleted by Bob we have an *update-delete* conflict.
- If Alice adds information to a fixed-point, which Bob did not, this is a *grow-left* conflict;
- If Bob adds information to a fixed-point, which Alice did not, 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 familiar from existing version control systems. We refer to these conflicts as *update* conflicts.

The *grow* conflicts are slightly more subtle, and in the majority of cases they can be resolved automatically. This class of conflicts roughly corresponds to the *alignment table* that *diff3* calculates [9] 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. Hence, we have no way of automatically knowing how this new information affects the rest of the file. This depends on the semantics of the specific file, therefore it is 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. As our patch data type is indexed by the types on which it operates, we can distinguish conflicts according to the types on which they may occur. For example, an *update-update* conflict must occur on a coproduct type, for it is the only type for which Patches over it can have different inhabitants. The other possible conflicts must happen on a fixed-point. In Agda, we can therefore define the following data type describing the different possible conflicts that may occur:

```
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

Although we now have fixed the data type used to represent conflicts, we still need to define our residual operator. As we discussed previously, the residual will return a new patch, but may fail in two possible ways. If the input patches are not aligned, we will return $\mathsf{nothing}$; if there is a conflict, we will record the precise location of the conflict using the $\mathsf{D\text{-}A}$ constructor. In this fashion, we have separate types to distinguish between patches without conflicts and patches arising from our residual computation containing unresolved conflicts.

The final type of our residual 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. Instead of displaying the entire Agda definition[1], we will discuss the key branches in some detail. We begin by describing the branch when one patch enters a fixed-point, but the other deletes it:

[1] The complete Agda code is publicly available and can be found in `omitted for double-blind review`.

```
res (Dμ-dwn dx :: dp) (Dμ-del y :: dq)
  with gapply dx y
...| nothing = nothing
...| just y' with y == y'
...| True  = res dp dq
...| False = _ :: _ (Dμ-A (UpdDel y' y)) <$> res dp dq
```

Here we are computing the residual:

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

We want to describe how to apply the changes $(\mathsf{D}\mu\text{-dwn}\ dx\ ::\ dp)$ to a structure that has been modified by the patch $(\mathsf{D}\mu\text{-del}\ y\ ::\ dq)$. Note that 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 these two patches are not aligned, and we simply return nothing. If we can apply dx to y, however, this will result in a new structure y'. We then need to compare y to y'. If they are different we have an update-delete conflict, signaled by $\mathsf{D}\mu\text{-A}\ \ (\mathsf{UpdDel}\ y'\ y)$. If they are equal, then dx is the identity patch, and no new changes were introduced. Hence we can simply suppress the deletion, $\mathsf{D}\mu\text{-del}$, and continue recursively.

Next, lets see how *grow* conflicts are detected by taking a look at the branches that handle insertion.

```
res (Dμ-ins x :: dp) (Dμ-ins y :: dq) with x == y
...| True  = _ :: _ (Dμ-dwn (cast (gdiff-id x))) <$> res dp dq
...| False = _ :: _ (Dμ-A (GrowLR x y)) <$> res dp dq
res dp (Dμ-ins x :: dq) = _ :: _ (Dμ-A (GrowR x)) <$> res dp dq
res (Dμ-ins x :: dp) dq = _ :: _ (Dμ-A (GrowL x)) <$> res dp dq
```

Whenever we find two overlapping insertions, we need to check whether the inserted values are equal. If they turn out to be different, we have a GrowLR conflict since we cannot decide if x should come before y or vice-versa. If they are equal, however, we just copy x, since we do not want to insert x twice, and continue recursively. The cast function will just cast a Patch to a PatchC without introducing any actual conflict. Unilateral insertions simply become GrowL and GrowR conflicts, that can later be easily solved if the context allows.

The remaining cases follow a similar reasoning. The residual function is defined by structural induction over patches. The diagonal cases are easy to solve. The off-diagonal cases require a similar reasoning, in p/q the idea is to come up with a patch, if possible, that can be applied to an object already modified by q but still produces the changes specified by p. When not possible we need to distinguish between two conflicting patches and two non-aligned patches.

The attentive reader might have noticed a symmetric structure on conflicts. This is no coincidence. In fact, we can prove that the residual of p/q have the same conflicts, modulo symmetry, as q/p.

The symmetric conflict of UpdDel x y is DelUpd y x; of GrowL x is GrowR x; etc. The function C-sym, which computes the symmetric of a conflict, respects the usual property C-sym $\cdot$ C-sym $\equiv$ id. The function conflicts extracts the list of conflicts of a PatchC. Hence, with a slight abuse of notation, we have that:

$$\mathsf{conflicts}\ (p/q) \equiv \mathsf{map}\ \mathsf{C\text{-}sym}\ (\mathsf{conflicts}\ (q/p))$$

This lemma provides further evidence that the usage of residuals or patch commutation (as proposed by some version control systems such as Darcs[1]) are not significantly different. This means that the p/q and q/p, although different, have symmetrical conflicts.

Merge Strategies When the residual operation manages to merge two patches, without introducing conflicts, we require no further user interaction. When the calculation of a residual does introduce conflicts, however, we need further information to eliminate these conflicts and produce a pair of conflict-free patches. Since the conflicts of one residual will be the symmetric of the conflicts of the other residual, we want to have a single *merge strategy* e to resolve them. We can visualize this in Figure 7.

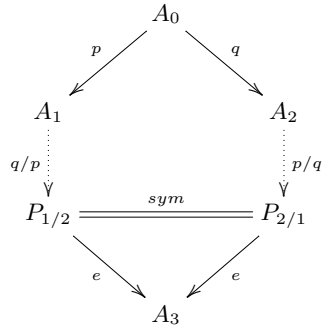

Figure 7. Residual patch square

The residual arrows are dotted since residuals do not produce Patches, but PatchC's, that is, there might be conflicts to be resolved. They cannot be applied yet. Therefore $P_{1/2}$ and $P_{2/1}$ have type PatchC, and should not be confused with objects that Alice and Bob can edit. Nevertheless we would like to find suitable conflict resolutions that allow us to extend q/p and p/q to yield conflict-free patches. What information do we need to resolve conflicts?

Assuming that q and p are aligned patches, we want e to have type $\mathsf{PatchC}\ A \to \mathsf{Patch}\ A$, that is, to solve conflicts! This type, however, would imply that e is total, therefore it could solve *all* conflicts. This is not very realistic. To have some more freedom we shall define a *merge strategy* to be a function of type $\mathsf{PatchC}\ A \to \mathsf{B}\ (\mathsf{Patch}\ A)$, for an arbitrary behavior monad B. An interactive merge strategy would have $\mathsf{B} = \mathsf{IO}$, a partial merge strategy would have $\mathsf{B} = \mathsf{Maybe}$, etc.

Hence the key question becomes: how to define e? As it turns out, we cannot completely answer that. The *merge strategy* e depends on the semantics of the files being diffed. The users should be the ones defining their own *merge strategies*, as they have the domain specific knowledge to do so. We must, however, provide them with the right tools for the job.

It is important to understand that this problem can be divided in two separate parts: (A) how do we traverse the PatchC structure and (B) how do we resolve the conflicts we find in the leaves.

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

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

In the future we would like to have a library of *solvers* and different traversals of a PatchC together with a calculus for them. Allowing one to prove lemmas about the behavior of some *merge strategies*, that is, a bunch of *solvers* combined using different traversals. More importantly, we are actively investigating how can one prove that a *merge strategy* converges. We conjecture that a sufficient condition for the convergence of a *merge strategy* can be extracted from the proof of the symmetric structure of residuals. This is left as future work. Nevertheless, our Haskell prototype already provides some non-trivial traversals and solvers

that converge for a large amount of practical cases, as we shall see in the next section.

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 demonstrating how these ideas may be implemented in a Haskell prototype, with an emphasis on its extended capability of handling non-trivial conflicts. A great advantage of our choice of type universe is that we it closely follows the traditional 'sums-of-products' view of Haskell data types, and can be readily transcribed to typeclasses in Haskell. The source code of our prototype is available online [2].

The central type class in our prototype is *Diffable*, that 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 sizeEIU function, presented in Section 2.4; *Patch* is a GADT [15] corresponding to 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) where
...
instance (Eq a, Eq b, Diffable a, Diffable b)
  ⇒ Diffable (Either a b) where
...
```

To handle fixed points, we need to provide the same plugging and unplugging functionality as in Agda. 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)
```

We then define a class and some template Haskell functionality to generate the 'sums-of-products' representation of a Haskell data type, which we can use as the input of our generic functions. 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
```

There is, however, one extension we need to be able to handle built-in types, such as *Char* or *Int*. 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 Copying and moving

In order to show the full potential of our approach, we will develop a simple example showing how one can define and run a new conflict resolution algorithm for arbitrary datatypes, capable of *copying* and *moving* subtrees. We will first introduce some simple definitions and then explore how refactoring can happen there.

Our case study will be centered on CSV files with integers on their cells. The canonical representation of this CSV format is given by the type T, defined below.

```
type T = List (List (Atom Int))
```

Note that the *List* type is defined as an element of our universe 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, and hence has a *HasAlg* and *HasSOP* instance. It is easy to see that T is isomorphic to $[[Int]]$. We will work with $[[Int]]$ in the following example for better readability.

We are now ready to go into our case study. Imagine both Alice and Bob clone a repository containing a single CSV file $l0 = [[1,2],[3]]$. 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. Using a slightly simplified notation, these two changes may be represented by the following two patches:

```
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 []]
```

[2] `Omitted for double-blind review`

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 conflict that may be resolved by copying some edited subtree. Moreover, from residual symmetric structure, 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 move we would expect the the resulting CSV to be $lR = [[2], [3, 12]]$. 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 deleted subtrees. If they match, we either copy or insert the *updated* version of the object.

Recall Section 3.1, where we described how conflicts may be resolved by a *merge strategy*, mapping conflicts to patches. Our Haskell prototype library already explores several different *merge strategies* that can be assembled to handle specific kinds of conflicts.

In the context provided by the current example, we will use a traversal $walkPWithCtx$ with a *solver* $sUpdCtx$.

$$\begin{aligned} sUpdCtx &:: Cf\ a \to [\forall\ x\ .\ Cf\ x] \to Maybe\ (Patch\ a) \\ walkPWithCtx &:: (Cf\ a \to [\forall\ x\ .\ Cf\ x] \to Maybe\ (Patch\ a)) \\ &\quad \to PatchC\ b \to PatchC\ b \end{aligned}$$

The $walkPWithCtx$ *traversal* will perform the aforementioned two traversals. The first one records the conflicts whereas the second one applies $sUpdCtx$ to the conflicts. The $sUpdCtx$ *solver*, in turn, will receive the list of all conflicts (context) and will try to match the *grow* operations with the *deletions*. Note that the *merge strategy* returns a $PatchC$, as the *merge strategy* is *partial*. It will leave the conflicts it cannot solve untouched. The 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 different *merge strategies* and a tool set to build custom ones, but since different domains will have different conflicts and conflict semantics, it is up to the user to program the best strategy for their particular domain.

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:

$$\begin{aligned} Just\ pAR &= resolved\ (walkPWithCtx\ sUpdCtx\ (pB \ / \ pA)) \\ Just\ pBR &= resolved\ (walkPWithCtx\ sUpdCtx\ (pA \ / \ pB)) \end{aligned}$$

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 $lR = [[2], [3, 12]]$ as a result.

As we can see from this example, our framework allows for a definition of different conflict resolution 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 a novel approach to version control systems, enhancing the diff and merge algorithms with information about the structure of data under control. We provided the theoretical foundations and created a Haskell prototype, demonstrating the viability of our approach. Our algorithms can be readily applied to any algebraic data type in Haskell, as these can all be represented in our type universe. We have also shown how this approach allows one to define custom conflict resolution strategies, such as those that attempt to recognize the copying of subtrees. The work of Lempsink et al. [11] and Vassena [19] are the most similar to our. We use a drastically different definition of patches. The immediate pros of our approach are the ability to have more freedom in defining conflict resolution strategies and a much simpler translation to Haskell. Our choice of universe, however, makes the development of proofs much harder.

There are several pieces of related work that we would like to mention here:

Antidiagonal Although easy to be confused with the diff problem, the antidiagonal is fundamentally different from the diff/apply specification. Piponi [16] defines the antidiagonal for a type T 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 [12], 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 6) 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 [1]. 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 [8]. 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.

Homotopical Patch Theory Homotopy Type Theory, and its notion of equality corresponding to paths in a suitable space, can also be used to model patches. Licata et al [3] developed such a model of patch theory.

Separation Logic Swierstra and Löh [17] use separation logic and Hoare calculus to be able to prove that certain patches do not overlap and, hence, can be merged. They provide increasingly more complicated models of a repository in which one can apply such reasoning. Our approach is more general in the file structures it can encode, but it might benefit significantly from using similar concepts.

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 Further work

Cost, Inverses and Lattices 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 the presence of equal cost alternatives, it does not favor any particular patch. This may lead to different, yet observationally equivalent results. In general, we do

not have much room to reason about the result of a gdiffL. A better definition of $\_\sqcup_\mu\_$, that can provide more properties is paramount for a more careful study of the cost function.

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. There is one drawback of our approach, that we would like to point out here. 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
```

Using ill-patch, we try to insert a suc constructor, which requires one recursive argument, but instead of providing this argument, the patch ends prematurely.

Ideally, we would like to make $\mathsf{D}\mu$ type-safe by construction, thereby ruling out such ill-formed patches. To do so, we revisit our definition of patches, adding two new indices of type $\mathbb{N}$.

```
data Dμ₂ (A : {n : ℕ} → Tel n → U n → Set)
   {n : ℕ}(t : Tel n)(ty : U (suc n))
: ℕ → ℕ → Set where
```

Then $\mathsf{D}\mu_2\ A\ t\ ty\ i\ j$ will model a patch over elements that expects exactly i child nodes and produces j new children. This is very similar to how type-safety is guaranteed in previous work by Lempsink et al. [11]. As the type of the children is known, the only additional information required as this pair of a natural numbers.

Insertions, $\mathsf{D}\mu\text{-ins}\ x$, (and symmetrically, deletions $\mathsf{D}\mu\text{-del}\ x$) are easy to adapt. We can compute the number of children we require from the head, x, that we are inserting (or deleting). Recursive changes, $\mathsf{D}\mu_2\text{-dwn}\ dx$, however, are more subtle. The easy fix would be to say that $\mathsf{D}\mu_2\text{-dwn}\ dx$ may never change a constructor, and hence it will not change its arity. However, for nested data types such as rose trees, this is condition is insufficient:

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

The arity of the constructor for rose trees will vary. More precisely, it will be equal to the length of the list of recursive rose trees. We therefore can have two rose trees, with the same head constructor, each applied to a different number of child nodes:

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

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

Fortunately, the insight is that the patch dx already has the information about the arity of both its source and destination elements. We should be able to extract this information from the patch and provide the correct indexes to $\mathsf{D}\mu_2\text{-dwn}\ dx$. Proving that the arity extracted from a patch corresponds to the arity of an element is not entirely straightforward. We would like to address this issue in further 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 a residual system in the term-rewriting sense [5, 18]. 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.

This paper has demonstrated that it is feasible to define generic merge and diff algorithms to improve version control of structured data. We can use our algorithms to create specialized revision control systems for virtually every imaginable file format – the only information we need to do so is a Haskell data type modeling the data under revision. These generic algorithms are more precise than the standard `diff` based tools, resulting in more accurate conflict information, and as a result, a better overall user experience.

References

[1] Darcs theory. `http://darcs.net/Theory`. Acessed: Feb 2016.

[2] T. Altenkirch, C. Mcbride, and P. Morris. Generic programming with dependent types. In *Spring School on Datatype Generic Programming*. Springer-Verlag, 2006.

[3] C. Angiuli, E. Morehouse, D. R. Licata, and R. Harper. Homotopical patch theory. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, pages 243–256, New York, NY, USA, 2014. ACM.

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

[5] M. Bezem, J. Klop, R. de Vrijer, and Terese. *Term Rewriting Systems*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2003.

[6] P. Bille. A survey on tree edit distance and related problems. *Theor. Comput. Sci*, 337:217–239, 2005.

[7] E. D. Demaine, S. Mozes, B. Rossman, and O. Weimann. An optimal decomposition algorithm for tree edit distance. In *Proceedings of the 34th International Colloquium on Automata, Languages and Programming (ICALP 2007)*, pages 146–157, Wroclaw, Poland, July 9–13 2007.

[8] J. Jacobson. A formalization of darcs patch theory using inverse semigroups. *Available from* `ftp://ftp.math.ucla.edu/pub/camreport/cam09-83.pdf`, 2009.

[9] S. Khanna, K. Kunal, and B. C. Pierce. A formal investigation of diff3. In *Proceedings of the 27th International Conference on Foundations of Software Technology and Theoretical Computer Science*, FSTTCS'07, pages 485–496, Berlin, Heidelberg, 2007. Springer-Verlag.

[10] P. N. Klein. Computing the edit-distance between unrooted ordered trees. In *Proceedings of the 6th Annual European Symposium on Algorithms*, ESA '98, pages 91–102, London, UK, UK, 1998. Springer-Verlag.

[11] E. Lempsink, S. Leather, and A. Löh. Type-safe diff for families of datatypes. In *Proceedings of the 2009 ACM SIGPLAN Workshop on Generic Programming*, WGP '09, pages 61–72, New York, NY, USA, 2009. ACM.

[12] S. Mimram and C. D. Giusto. A categorical theory of patches. *CoRR*, abs/1311.3903, 2013.

[13] N. Mitchell. *Transformation and Analysis of Functional Programs*. PhD thesis, University of York, June 2008.

[14] U. Norell. Dependently typed programming in agda. In *Proceedings of the 4th International Workshop on Types in Language Design and Implementation*, TLDI '09, pages 1–2, New York, NY, USA, 2009. ACM.

[15] S. Peyton Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for gadts. *SIGPLAN Not.*, 41(9):50–61, Sept. 2006.

[16] D. Piponi. The antidiagonal. `http://blog.sigfpe.com/2007/09/type-of-distinct-pairs.html`, 2007. Acessed: Feb 2016.

[17] W. Swierstra and A. Löh. The semantics of version control. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! '14, pages 43–54, 2014.

[18] S. Tieleman. Formalisation of version control with an emphasis on tree-structured data. Master's thesis, Universiteit Utrecht, Aug. 2006.

[19] M. Vassena. Svc, a prototype of a structure-aware version control system. Master's thesis, Universiteit Utrecht, 2015.