Structure-aware version control

A generic approach using Agda

DRAFT

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, which may lead 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; Our ideas can be, nevertheless, be transcribed to Haskell yielding 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 to observe unnecessary conflicts in many situations.

For example, consider the following example CSV file that records the marks, unique identification numbers, and names three of 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 [17]. We have chosen a universe that closely resembles the algebraic data types that are definable in functional languages such as Haskell (Section 2.1). By being able to *diff* any Haskell datatype, we can in particular *diff* the output of any Haskell parser.
- We define generic algorithms for computing and applying a diff and prove that these algorithms satisfy several basic correctness properties (Section 3.3).
- 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 sets the ground for resolving conflicts (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 [8], improving the work of Klein [12]. The instantiation of this problem to lists is known as the *least common subsequence* (LCS) problem [5, 7]. 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 *number of operations*, but instead choose a cost model that more accurately captures which changes are important to the specific data type in question. In practice, the *diff* tool creates patches by observing changes on a line-by-line basis. However, when different changes must be merged, using tools such as *diff3* [11], there is room for improvement.

1.1 Patches, informally

Before we delve into the definition of *patches*, we first have to specify what *patches* are supposed to be. Intuitively, a *patch* is simply the description of a transformation between two *values* of the same type.

The usual operations one expects to perform over *patches* are: (A) given two *values*, we need to be able to describe how to transform one into the other, and, (B) given a *patch* and a *value*, we need to be able to apply this *patch* to the *value*, if possible.

From this description, we could already define a trivial patch over any type A equipped with decidable equality, which indeed have the expected operations: (A) a *diff* function; and (B) an *apply* function.

```
Patch : Set
Patch = A × A

diff : A → A → Patch
diff x y = (x , y)

apply : Patch → A → Maybe A
apply (x , y) z with x == z
...| True  = just y
...| False = nothing
```

It should be clear that this implementation of patches is not desirable. Even though creating a patch is very efficient, the resulting patches do not tell us anything about which changes have been made. Our specification should rule out this trivial implementation. In particular, we expect a few more properties of *patches*:

i) They should describe the *minimal* transformation between two *values*, for some notion of minimality.

ii) Computing and applying patches must be efficient.

Nevertheless, every patch must store information about its source on which it operates and the target value it produces. The dummy implementation above, however, stores too much information. We will show how to exploit A's structure to address this. Before we present the data type generic definitions and algorithms, however, we will present a specific instance of our diff algorithm for binary trees.

1.2 Diffing Binary Trees

On this section we will define a *patch* for binary trees together with its *diff* function. For the purpose of this example, we assume the existence of a Patch, diffA and costA for diffing the elements of type A inside the tree.

```
data Tree (A : Set) : Set where
  Leaf : Tree A
  Node : A → Tree A → Tree A → Tree A
```

The first step is to fix a t : Tree A and figure out the possible structural transformations one can perform over t. As this is the information we need to represent using a *Patch*. For this situation:

i) We can add or remove subtrees from t.

ii) If t is a Node with a value $a : A$ inside, we can modify a and recursively *diff* the two subtrees of t.

To calculate a patch between two trees, we need to find a way of traversing recursive types, inserting and removing values as we go. We begin by observing that the type of binary trees is, in fact, the least fixpoint of a (bi)functor:

```
TreeF : Set → Set → Set
TreeF A X = Unit ⊎ A × X × X

Tree : Set → Set
Tree A = Fix (TreeF A)
```

We then define the type of the *head* of a Tree to be isomorphic to TreeF A 1, where 1 is the unit type. The *head* of a fixpoint gives us information about which constructor, together with non-recursive arguments, is used as the topmost constructor in a value. It is not hard to see that TreeF A $1 \approx$ Maybe A.

Although this specific example is around binary trees, the general case has to handle the fixpoint of any functor (definable in our choice of universe, of course). The idea is compute an alternative representation of the values of a fixpoint. The very definition of a fixpoint says that the values of a Fix F will be composed of a constructor, some non-recursive and some recursive parts. We define *head* and *children* of a fixpoint to access these respective parts.

For the present example, we can always represent a Tree A in a list of TreeF A 1, by adding the *head* of the current value to the beginning of the list and recursing on the *children*. We call this *serialization*.

```
hd : {A : Set} → Tree A → Maybe A
hd Leaf          = nothing
hd (Node x _ _)  = just x

ch : {A : Set} → Tree A → List (Tree A)
ch Leaf          = []
ch (Node _ l r)  = l :: r :: []
```

The serialization transforms a Tree into a list of things that describe the *shape* of the tree as seen by traversing its nodes in a given order, and can later be used to reconstruct the Tree. Now we just need to be able to insert and delete *heads* in our serialized tree.

```
serialize : {A : Set} → Tree A → List (Maybe A)
serialize t = hd t :: concat (map serialize (ch t))
```

In short, a serialized Tree A, or, List (TreeF A 1), can be seen as the list of constructors used as they are seen in a preorder traversal of the Tree.

By reducing a tree to a list, or, any fixpoint into a list of *heads* for that matter, the definition of patches becomes simpler. The structural operations one can perform over lists are: copy an empty list; insert or delete a *head* from the beginning of the list and recurse on the tail; or modify the *head* in the beginning of the list and recurse on the tail. Encoding this in a datatype gives us:

```
data TPatch (A : Set) : Set where
  Nil  :  TPatch A
  Ins  : Maybe A →     TPatch A → TPatch A
  Del  : Maybe A →     TPatch A → TPatch A
  Mod  : Patch (Maybe A)
       → TPatch A → TPatch A
```

With a representation of the possible transformations an element of Tree A can undergo we are ready to write our first *diffing* algorithm. Note how we will diff lists of trees and serialize them as we proceed, instead of serializing everything first. This is mainly an efficiency concern.

```
diff : {A : Set} → (as bs : List (Tree A)) → TPatch A
diff []        []        = Nil
diff (x :: xs) []        = Del (hd x) (diff (ch x ++ xs) [])
diff []        (y :: ys) = Ins (hd y) (diff [] (ch y ++ ys))
diff (x :: xs) (y :: ys)
  = let
      d1 = Ins  (hd y) (diff (x :: xs) (ch y ++ ys))
      d2 = Del  (hd x) (diff (ch x ++ xs) (y :: ys))
      d3 = Mod  (diffA (hd x) (hd y))
                (diff (ch x ++ xs) (ch y ++ ys))
    in d1 ⊔ d2 ⊔ d3
```

The three base cases are not very interesting, if one of the arguments is the empty list, there is only so much one can do. The last case is slightly more complicated. We can always delete or insert a Maybe$A$, but now, additionally, we can also compare the Maybe$A$ values on the beginning of both lists and try to change one into the other. This is done by the diffA function. Afterwards, we have to choose one of the three patches we have: d_1, d_2 and d_3. The associative operator _ ⊔ _ simply chooses the patch with the least *cost*.

Consider the situation in which a Leaf is transformed into a Node x, for some $x : A$. There are two ways for performing this transformation. We can Del the current hd Leaf and Ins the hd (Node x), this patch would be encoded by:

$$\text{Del nothing (Ins (just } x\text{) Nil)} \qquad \text{(p.1)}$$

Or, we could Mod the constructor from a Leaf into a Node x:

$$\text{Mod (diffA nothing (just } x\text{)) Nil} \qquad \text{(p.2)}$$

The cost function is the tool we use to favor some patches over others. In this example, which of the two should we prefer?

It is clear that the patch p.1 should be selected, as it immediately tells us that the *structure* of the tree will change, by deletions and insertions. Whereas the second patch, p.2, gives the impression that we are simply changing the value inside a Node. That is, patch p.1 describes the actual changes better than patch p.2. Hence, patch p.1 should have a lower cost.

When we say we want patches to be minimal, we are referring to them having a minimal cost. Thus, the cost notion should express how closely a patch represents the changes in a descriptive fashion instead of the computational effort needed to apply such patch. We will define this function for the general case later on, in Section 3.4.

Applying *patches* is simple: we traverse the patch structure and update the tree that is being patched as we go along. Crucially, it relies on the plug function to reassemble trees from their head and children. In this example, we can define the plug function as follows:

```
plug : {A : Set}
  → Maybe A → List (Tree A)
  → Maybe (Tree A)
plug nothing  ts           = just Leaf
plug (just x) (l :: r :: ts) = just (Node x l r)
plug _        _            = nothing
```

Note that the apply function has to be partial, for the same reason that plug is partial: if we are *plugging* a just, we need at least two Trees. This is not a problem as we can prove that the *patches* produced and manipulated by our algorithms are *well-formed* and applying them will always produce a valid result.

2. Generic Programming

Now that we have an intuition of what patches should be like, and what sort of functions we need to define them, we need to introduce some *generic programming* notions in order to solve the problem in the general case. As usual, we start by choosing our universe of types. We have chosen to define patches on the universe of *Regular Tree Types* [16], as it contains most of the algebraic data types one can define in Haskell. We will give a brief overview of the universe; a complete library for generic programming can be found online [1].

2.1 Regular Tree Types

The universe of regular tree types [16] (sometimes also called context-free types [3]) defines a set of *codes* and an interpretation function from *codes* to Set. This universe can express polynomial types with type application and least fixpoints.

The type of *codes* with n (de Bruijn style) type variables 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
  var   : {n : ℕ} → U (suc n)
  wk    : {n : ℕ} → U n → U (suc n)
```

The ℕ index gives the number of free type variables available in the expression. The most recently bound variable may be referred to using the var constructor; the weakening constructor wk discards the topmost variable, allowing access to the others. The least fixpoint, μ , and definitions, def , bind a variable. Products, coproducts, the unit type and the empty type are standard.

As a simple example, we can represent the type of binary trees of booleans as:

```
boolU  : U 0
boolU  = u1 ⊕ u1

treeU  : U 1
treeU  = μ (u1 ⊕ (wk var ⊗ var ⊗ var))

btreeU : U 0
btreeU = def treeU boolU
```

Here we use the def constructor to instantiate the treeU type.

We now need to provide an interpretation function that maps a given code, in U, to a Set. On a first try, it would be natural to attempt interpreting only *closed* type expressions, U 0, using explicit substitution whenever necessary. This approach, however, would require some non-trivial substitution machinery [2], and complicate the definition of our generic operations. Instead, we choose to interpret open type expressions in a suitable environment.

We could choose the environment to be a list of types, describing how to interpret every de Bruijn index. In our scenario, however, it needs to be a *telescope* [9]. That is, every new variable may refer to previous variables in its definition.

```
data T : ℕ → Set where
  []   : T 0
  _::_ : {n : ℕ} → U n → T n → T (suc n)
```

[1] https://github.com/VictorCMiraldo/cf-agda

With codes and telescopes at hand, we can interpret every type expression without the need for explicit substitutions or renamings. For every *code* T and every *telescope* Γ, we can compute a set $[\![T]\!]_\Gamma$ as follows:

$$\begin{aligned}
[\![\mathsf{u0}]\!]_\Gamma &= 0\\
[\![\mathsf{u1}]\!]_\Gamma &= 1\\
[\![T_a \oplus T_b]\!]_\Gamma &= [\![T_a]\!]_\Gamma + [\![T_b]\!]_\Gamma\\
[\![T_a \otimes T_b]\!]_\Gamma &= [\![T_a]\!]_\Gamma \times [\![T_b]\!]_\Gamma\\
[\![\mathsf{def}\ F\ x]\!]_\Gamma &= [\![F]\!]_{x,\Gamma}\\
[\![\mathsf{var}\ ]\!]_{x,\Gamma} &= [\![x]\!]_\Gamma\\
[\![\mathsf{wk}\ T]\!]_{x,\Gamma} &= [\![T]\!]_\Gamma\\
[\![\mu\ T]\!]_\Gamma &= [\![T]\!]_{\mu\ T,\Gamma}
\end{aligned}$$

We will define this interpretation as an Agda datatype.

```
data ElU : {n : ℕ} → U n → T n → Set where
  unit  : {n : ℕ}{t : T n}
        → ElU u1 t
  inl   : {n : ℕ}{t : T n}{a b : U n}
        (x : ElU a t) → ElU (a ⊕ b) t
  inr   : {n : ℕ}{t : T n}{a b : U n}
        (x : ElU b t) → ElU (a ⊕ b) t
  _,_   : {n : ℕ}{t : T n}{a b : U n}
        → ElU a t → ElU b t → ElU (a ⊗ b) t
  top   : {n : ℕ}{t : T n}{a : U n}
        → ElU a t → ElU var (a :: t)
  pop   : {n : ℕ}{t : T n}{a b : U n}
        → ElU b t → ElU (wk b) (a :: t)
  mu    : {n : ℕ}{t : T n}{a : U (suc n)}
        → ElU a (μ a :: t) → ElU (μ a) t
  red   : {n : ℕ}{t : T n}{F : U (suc n)}{x : U n}
        → ElU F (x :: t)
        → ElU (def F x) t
```

Our universe of *codes* gives us a clear inductive structure that we can use to define generic functions. To improve readability of our code, we will sometimes drop Agda-specific syntax from now on, and instead, sketch the main ideas underlying our definitions. The complete development is available online at `https://github.com/VictorCMiraldo/cf-agda`.

Following the lines of the example, Section 1.2, the generic functions we will need throughout the paper are the generic versions of the *head*, *children* and *plug* functions. From now on, we assume we have these functions with the following types:

```
μ-hd   : [[ μ ty ]] t → [[ ty ]] (u1 :: t)
μ-ch   : [[ μ ty ]] t → List ([[ μ ty ]] t)
μ-plug : [[ ty ]] (u1 :: t) → List ([[ μ ty ]] t)
   → Maybe ([[ μ ty ]] t)
```

Moreover, plug must satisfy the expected correctness property:

$$\forall x\ .\ \mathsf{plug}(\mathsf{hd}\ x)(\mathsf{ch}\ x) \equiv \mathsf{just}\ x$$

We stress that the implementation of the aforementioned functions is slightly different, and requires a more general type. The complete definitions can be found in our library.

3. Structural Patches

Following the inductive structure given by our *codes*, we shall define the type of *patches* over a given type.

Recalling Section 1.1, the idea is using as much (type) structure as possible to mimic our simple definition of patches, as a pair of source and target. More formally, our patch type should behave as the diagonal functor Δ mapping an object A to the pair (A, A) with analogous action on arrows.

In this section we will define $\mathsf{Patch}_\Gamma\ T$, the type of patches over some code T and telescope Γ. The subscripts Γ will be omitted when they can be inferred by the context. We will use $=$ to refer to definitions, $\equiv$ to refer to propositional equality and $\approx$ to refer to isomorphism.

Let us start by defining patches over the most basic types in our universe.

$T \equiv$ ***u0*** ***;*** When T is the empty type, the type of patches is on T is empty. There are no transformations one can make because there are no values to be transformed.

$$\mathsf{Patch}\ \mathsf{u0} = 0 \approx \Delta[\![\mathsf{u0}]\!]$$

$T \equiv$ ***u1*** ***;*** When T is the unit type, there is only one possible transformation: no change at all.

$$\mathsf{Patch}\ \mathsf{u1} = 1 \approx \Delta[\![\mathsf{u1}]\!]$$

$T \equiv T_a \otimes T_b$ ***;*** When T is a product of two types, again, there is only one possible transformation: to transform the components of the pair separately:

$$\begin{aligned}
\mathsf{Patch}\ (T_a \otimes T_b) &= \mathsf{Patch}\ T_a \times \mathsf{Patch}\ T_b\\
&\approx \Delta[\![T_a]\!] \times \Delta[\![T_b]\!]\\
&\approx \Delta[\![T_a \otimes T_b]\!]
\end{aligned}$$

$T \equiv T_a \oplus T_b$ ***;*** When T is a coproduct of two types, we are faced with more options. There are four possibilities: one for each choice of inl and inr for the source and target. When tag associated with the source and target coincide, the patch only needs information about the underlying change. When the tag associated with the source and target is different, the patch on the coproduct should record both.

$$\begin{aligned}
\mathsf{Patch}\ (T_a \oplus T_b) &= \mathsf{Patch}\ T_a + \mathsf{Patch}\ T_b + 2 \times [\![T_a]\!] \times [\![T_b]\!]\\
&\approx \Delta[\![T_a]\!] + 2 \times [\![T_a]\!] \times [\![T_b]\!] + \Delta[\![T_b]\!]\\
&\approx \Delta[\![T_a \oplus T_b]\!]
\end{aligned}$$

The universe of context free types uses a *telescope* to interpret variables and application. In fact, if we look closely at the definition of ElU for var , wk and def we can see that all we need to do is manipulate the *telescope*. The definition of Patch for these constructors will follow the same approach.

$T \equiv$ ***var*** ***;*** When T is the *topmost* variable, we can assert that we have at least one element on Γ, hence $\Gamma = \tau, \Gamma'$.

$$\begin{aligned}
\mathsf{Patch}_{\tau,\Gamma'}\ \mathsf{var} &= \mathsf{Patch}_{\Gamma'}\ \tau\\
&\approx \Delta[\![\tau]\!]_{\Gamma'}\\
&\approx \Delta[\![\mathsf{var}\ ]\!]_{\tau,\Gamma'}
\end{aligned}$$

$T \equiv$ ***wk*** T ***;*** Weakenings are also very simple, we just need to drop the topmost variable and Patch recursively. Here, we also have a non-empty *telescope*, hence $\Gamma = \tau, \Gamma'$.

$$\begin{aligned}
\mathsf{Patch}_{\tau,\Gamma'}\ (\mathsf{wk}\ T) &= \mathsf{Patch}_{\Gamma'}\ T\\
&\approx \Delta[\![T]\!]_{\Gamma'}\\
&\approx \Delta[\![\mathsf{wk}\ T]\!]_{\tau,\Gamma'}
\end{aligned}$$

$T \equiv$ ***def*** $F\ x$ ***;*** When $T = \mathsf{def}\ F\ x$, we simply need to patch F, adding x to the telescope in order to bind the topmost variable, that is, de Bruijn index 0, of F to x.

$$\begin{aligned} \mathsf{Patch}_\Gamma\ (F\ x) &= \mathsf{Patch}_{x,\Gamma}\ F \\ &\approx \Delta[\![F]\!]_{x,\Gamma} \\ &\approx \Delta[\![\mathsf{def}\ F\ x]\!]_\Gamma \end{aligned}$$

3.1 Least Fixpoints

Handling finite types with variables and application is just routine induction. Patching fixpoints is more challenging as they can *grow* and *shrink* arbitrarily. That is, we can always insert and delete subtrees.

To give a generic definition, we need to find a way to uniformly describe how the fixpoints in our universe *grow* or *shrink*. The idea is that the fixpoint of any F-structure can be serialized as a list of $F1$ by fixing a traversal order. This is a generalization of how we handled binary trees in Section 1.2. In fact, the generic serialization function can be defined as:

```
serialize : {n : ℕ}{t : T n}{ty : U (suc n)}
  → ElU (μ ty) t → List (ElU ty (u1 :: t))
serialize x = μ-hd x :: concat (map serialize (μ-ch x))
```

This gives us a uniform way to handle fixpoints generically. Following the same intuition from the patches over trees, Section 1.2, we can always insert or delete *heads* in the serialized fixpoint or modify the contents of a *head* recursively. Thus,

$$\mathsf{Patch}\ (\mu\ F)\ =\ \mathsf{List}(F\ 1 + F\ 1 + \mathsf{Patch}(F\ 1))$$

This reads as "A *patch* of the (least) fixpoint of an F-structure is a list of *edit operations* over $F\ 1$". Whereas the *edit operations* are, in turn, a coproduct representing insertion, deletion or modification, respectively.

But when we try to define a deserialization function, we run into problems. Take, for instance, the deserialization of the empty list. What should that be? The inverse of serialization is clearly a partial function.

Hence, it is clear that if we use this serialization-based approach, our definition of $\mathsf{Patch}\ \ (\mu\ \ F)$ is *not* isomorphic to $\Delta\ (\mu F)$, precisely because of the partiality of deserialization.

We could define $\mathsf{Patch}\ (\mu\ F)$ a bit more carefully. The use of indexed lists to keep track of how many elements a patch consumes and produces or the use of Σ-types to restrict the patches to those that have a well defined *source* and a *destination* could do the job. The actual implementation uses the Σ-type approach, but for presentation and simplicity purposes, we will omit this for now.

3.2 Patches, in Agda

With a general idea of patches at hand, we can now define the Agda datatype of *patches* by induction on *codes* and *telescopes*.

We will define the type $\mathsf{D}\ \ A\ \ t\ \ ty$ of *diffs* for the code ty and telescope t with a free-monad structure on A. This parameter A is used to add information, as we shall see shortly; its type, $TU{\rightarrow}Set$, is just the type of inductive type-families over *codes* and *telescopes*, defined by $\forall\{n\} \to \mathsf{T}\ n \to \mathsf{U}\ n \to \mathsf{Set}$.

```
data D (A : TU→Set)
     : {n : ℕ} → T n → U n → Set
where
  D-unit  : {n : ℕ}{t : T n} → D A t u1

  D-pair  : {n : ℕ}{t : T n}{a b : U n}
          → D A t a → D A t b → D A t (a ⊗ b)
  D-inl   : {n : ℕ}{t : T n}{a b : U n}
          → D A t a → D A t (a ⊕ b)
  D-inr   : {n : ℕ}{t : T n}{a b : U n}
          → D A t b → D A t (a ⊕ b)
  D-setl  : {n : ℕ}{t : T n}{a b : U n}
          → ElU a t → ElU b t → D A t (a ⊕ b)
  D-setr  : {n : ℕ}{t : T n}{a b : U n}
          → ElU b t → ElU a t → D A t (a ⊕ b)

  D-def   : {n : ℕ}{t : T n}{F : U (suc n)}{x : U n}
          → D A (x :: t) F → D A t (def F x)
  D-top   : {n : ℕ}{t : T n}{a : U n}
          → D A t a → D A (a :: t) var
  D-pop   : {n : ℕ}{t : T n}{a b : U n}
          → D A t b → D A (a :: t) (wk b)

  D-A  : {n : ℕ}{t : T n}{ty : U n}
       → A t ty → D A t ty

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

Besides the definitions for the basic type constructors, as we presented previously, the D-A constructor can be used to store values of type A. As a result, the type for diffs forms a free monad by construction. This structure will be used for storing additional information, when we have conflicts, as we shall see later (Section 4.1).

The only other interesting case is that for fixed points. These are handled by a list of *edit operations*:

```
data Dμ (A : TU→Set)
     : {n : ℕ} → T n → U (suc n) → Set
where
  Dμ-ins  : {n : ℕ}{t : T n}{a : U (suc n)}
          → ElU a (u1 :: t) → Dμ A t a
  Dμ-del  : {n : ℕ}{t : T n}{a : U (suc n)}
          → ElU a (u1 :: t) → Dμ A t a
  Dμ-dwn  : {n : ℕ}{t : T n}{a : U (suc n)}
          → D A (u1 :: t) a → Dμ A t a

  Dμ-A  : {n : ℕ}{t : T n}{a : U (suc n)}
        → A t (μ a) → Dμ A t a
```

In addition to the constructors for inserting, deleting, or modifying subtrees, we add a new constructor storing the parameter A.

Finally, we define the type synonym $\mathsf{Patch}\ t\ ty$ as $\mathsf{D}\ (\lambda\ \_\ \_ \to \bot)\ t\ ty$. In other words, a Patch is a D structure that never uses the D-A constructor, that is, has no extra information.

Source and Destination From the first sections of the paper we have been stressing that we want our patches to be isomorphic to a pair of values, representing the patch's *source* and a *destination*. As you might expect, we can compute these values from any given patch:

```
D-src : {A : TU→Set}{n : ℕ}{t : T n}{ty : U n}
  → D A t ty → Maybe (ElU ty t)

D-dst : {A : TU→Set}{n : ℕ}{t : T n}{ty : U n}
  → D A t ty → Maybe (ElU ty t)
```

Note that these functions are partial. There are some pathological cases in which these may fail, precisely those that bump into the deserialization problem we mentioned earlier. There are two options for ruling out problematic patches from the elements of D. Firstly, we could use derivatives instead of *heads* for inserting and deleting subtrees, hence guaranteeing that they all have one hole. Alternatively, we could choose to add two additional $\mathbb{N}$ indexes to Dμ, keeping track of how many elements that patch expects and produces. Both these options complicate the further development considerably. We chose to let D represent more patches than we need and rule out the pathological cases using Σ-types, whenever necessary.

We then say that a Patch p is *well-formed* iff there exists two elements x and y such that D-src $p \equiv$ just x and D-dst $p \equiv$ just y. In Agda, we can define a data type expressing when a patch is well-formed as follows:

```
WF : {A : TU→Set}{n : ℕ}{t : T n}{ty : U n}
  → D A t ty → Set
WF {A} {n} {t} {ty} p
  = Σ (ElU ty t × ElU ty t)
    (λ xy → D-src p ≡ just (p1 xy) × D-dst p ≡ just (p2 xy))
```

It is mechanical to prove that eliminating constructors of D and Dμ preserve *well-formed* patches, which allows one to define functions by induction on *well-formed* patches only. This allows us to rule out any pathological examples in our developments.

3.3 Producing Patches

We are now ready to define a generic function gdiff that, given two elements of a regular tree type, computes the patch recording their differences. For finite types and type variables, the gdiff functions follows the structure of the type in an almost trivial fashion.

```
gdiff : {n : ℕ}{t : T n}{ty : U n}
  → ElU ty t → ElU ty t → Patch t ty
gdiff {ty = u1}        unit unit
  = D-unit
gdiff {ty = var}       (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 = 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 :: []))
```

Diffing fixpoints is much more challenging. Since we never really know how many children will need to be handled in each step, gdiffL handles lists of subtrees, or *forests*. Our algorithm, heavily inspired by [13], works as follows:

```
gdiffL : {n : ℕ}{t : T n}{ty : U (suc n)}
  → List (ElU (μ ty) t) → List (ElU (μ ty) t) → Patchμ t ty
gdiffL [] [] = []
gdiffL [] (y :: ys)
  = Dμ-ins (μ-hd y) :: (gdiffL [] (μ-ch y ++ ys))
gdiffL (x :: xs) []
  = Dμ-del (μ-hd x) :: (gdiffL (μ-ch x ++ 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
```

Here, μ-open x computes the pair of the head, μ-hd x and children μ-ch x of any given tree x.

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 finally, 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*. The associative operator $\_ \sqcup_\mu \_$ returns the patch with the lowest *cost*. As we shall see in section 3.4, this notion of cost is very delicate. Before we explore the *cost* function, however, let us introduce a few interesting results and special patches.

Correctness of gdiff As we mentioned previously, not all patches are well-formed. We can prove, however, that gdiff is guaranteed to produce *well-formed* patches:

$$\begin{aligned}\text{D-src}\ (\text{gdiff}\ x\ y) &\equiv \text{just}\ x\\ \text{D-dst}\ (\text{gdiff}\ x\ y) &\equiv \text{just}\ y\end{aligned}$$

Identity Patch For all $x : [\![ty]\!]_\Gamma$, we can compute the identity patch on x, written D-id x. Moreover, it has x as its *source* and *destination*.

In fact, looking at the definition of gdiff, it is not hard to see that whenever $x \equiv y$, gdiff x y will produce a patch without any occurrence of D-setl, D-setr, Dμ-ins and Dμ-del, as they are the only constructors that introduce *new information*. We call these constructors the *change-introduction* constructors.

Inverse Patch Given a patch p : Patch$_\Gamma$ ty, if it is not the identity patch, then it has some *change-introduction* constructors inside. We can compute the inverse patch of p, D-inv p by swapping D-setl's with D-setr's and Dμ-ins's with Dμ-del's. It satisfies the following properties:

$$\begin{aligned}\text{D-src}\ (\text{D-inv}\ p) &\equiv \text{D-dst}\ p\\ \text{D-dst}\ (\text{D-inv}\ p) &\equiv \text{D-src}\ p\end{aligned}$$

Therefore, if p is *well-formed*, then D-inv p is *well-formed*.

Composition of Patches Given two *well-formed* patches p, q : Patch$_\Gamma$ ty, if D-src $p \equiv$ D-dst q then we can define the composition of p and q, $p \circ_D q$, which also satisfies the expected properties:

$$\begin{aligned}\text{D-src}\ (p \circ_D q) &\equiv \text{D-src}\ q\\ \text{D-dst}\ (p \circ_D q) &\equiv \text{D-dst}\ p\end{aligned}$$

3.4 The Cost Function

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

```
cost : {n : ℕ}{t : T n}{ty : U n} → Patch t ty → ℕ
```

The cost of transforming x into y intuitively leads one to think about *how far is x from y*. We believe that the cost of a patch induce a *metric* on our universe:

```
dist x y = cost (gdiff x y)
```

Remember that we call a function *dist* a *metric* if the following three properties are satisfied:

$$
\begin{aligned}
dist\ x\ y = 0 \quad &\Leftrightarrow \quad x = y && (1)\\
dist\ x\ y \quad &= \quad dist\ y\ x && (2)\\
dist\ x\ y + dist\ y\ z \quad &\geqslant \quad dist\ x\ z && (3)
\end{aligned}
$$

We can now proceed to calculate the cost function from this specification.

Equation (1) tells that the cost of not changing anything must be 0, therefore, the cost of D-id x should be 0, for all x. That is easy to achieve, as D-id x is the patch over x with no *change-introduction* constructors, we just assign a cost of 0 to every non-*change-introduction* constructor.

Equation (2), on the other hand, tells us 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 be 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 will be 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). For now this suffices. We shall give them a concrete definition later on.

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. 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, $\#($gdiff $x\ y) + \#($gdiff $y\ z) = \#($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 $\#($gdiff $x\ y) + \#($gdiff $y\ z) > \#($gdiff $x\ z)$, and since the *change-introduction* constructors are the ones with non-zero cost, this also implies equation (3).

Let us make a short summary of what happened so far. We began by defining patches and how to compute them. We then saw the need of a relation over patches, that would let one choose between patches with the same source and destination. This motivates the cost function. In order to define the cost function, however, we started from its specification and computed a suitable (abstract) definition for cost. Given the special patches (identity, inverse and composition) and the restrictions imposed by the specification, we saw that there were only two values left to be defined, and for nearly whatever definition we gave to those values the cost will induce a metric.

Let costL $= sum \cdot map$ costμ, the cost function is then defined by:

```
cost (D-A ())
cost  D-unit        = 0
cost (D-inl d)      = cost d
cost (D-inr d)      = cost d
cost (D-setl xa xb) = c⊕ xa xb
cost (D-setr xa xb) = c⊕ xa 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)   = cμ x
costμ (Dμ-del x)   = cμ x
costμ (Dμ-dwn x)   = cost x
```

In order fill in the gaps that are left in the Agda code we abstract away $c_\oplus$ and c_μ, package everything inside a record and write the rest of the code passing those records as module parameters.

```
record Cost : Set where
  constructor cost-rec
  field
    c⊕  : {n : ℕ}{t : T n}{x y : U n}
        → ElU x t → ElU y t → ℕ
    cμ  : {n : ℕ}{t : T n}{x : U (suc n)}
        → ElU x (u1 :: t) → ℕ

    c⊕-sym-lemma  : {n : ℕ}{t : T n}{x y : U n}
                  → (ex : ElU x t)(ey : ElU y t)
                  → c⊕ ex ey ≡ c⊕ ey ex
```

It is straightforward to prove that the cost (D-id x) $\equiv 0$ and cost (D-inv p) $\equiv$ cost p. For the later we need the symmetry lemma over $c_\oplus$, which is why it is packaged together.

To complete our definition and be able to run our algorithm, we still need to choose suitable definitions for $c_\oplus$ and c_μ. Different cost models will favor certain changes over others – yielding very different behavior for our diff algorithm.

We will now calculate one possible choice for c_μ and $c_\oplus$ that favors 'smaller' changes further down in the tree. 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, in a CSV file context, this would consider inserting a new line to be a more expensive operation than updating a single cell.

The rest of this section is quite technical and might not be of much interest to some readers. In the end of the calculation we provide the definitions we use for $c_\oplus$ and c_μ in order to get the behavior we want. Nevertheless, let us 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.

```
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)
```

For now, we will only compare d_1 and d_3. Since the cost of inserting and deleting subtrees is necessarily 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 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:

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

Hence, cost d_1 is $c_\mu\ hdX + c_\mu\ hdY + w$, for $w =$ cost (tail d_3). Here hdX and hdY are values of the same type, ElU ty (tcons u1 t).

As our data types will typically be sums-of-products, hdX and hdY are values of the same finitary coproduct, corresponding to the constructors of a (recursive) data type.

We will now consider the patch redundancy problem we briefly mentioned in Section 1.2. Recall the two patches that could change a Leaf into a Node:

$(p.1)$ Del nothing (Ins (just x) Nil)
$(p.2)$ Mod (diffA nothing (just x)) Nil

As we mentioned on the example, the cost function is what is going to favor one over the other. Let us take a look at this very situation but in a more general setting. 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 D-setl $x'\ y'$ or a D-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 calculating the specification our functions c_μ and $c_\oplus$ needs to satisfy in order to obtain the desired behavior.

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

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 cost (gdiff $(i_j\ x')\ (i_j\ y')) + w$. Since gdiff $(i_j\ x')\ (i_j\ y')$ will reduce to gdiff $x'\ y'$ preceded by a sequence of D-inr and D-inr, which have zero cost. Hence, cost $d_3 =$ cost (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:

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

Recall that our objective was to calculate a specification for the cost function that guarantees as many constructors as possible are preserved. 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. 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.

The choice of c_μ and $c_\oplus$ 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. Nevertheless, since our algorithms are defined abstractly on the Cost details, we plan to later allow customization of the algorithm's behavior by changing the cost assigned to specific datatypes.

To run our diff algorithm, we define a generic sizeElU function and declare a *top-down* Cost as follows:

```
sizeElU : {n : ℕ}{t : T 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

top-down-cost
  = cost-rec (λ ex ey → sizeElU ex + sizeElU ey)
    sizeElU
    (λ ex ey → (+-comm (sizeElU ex) (sizeElU ey)))
```

3.5 Applying Patches

We have defined an algorithm to *compute* a patch, but we have not yet defined an algorithm to *apply* a patch. This is one of the simplest algorithms of our whole development. We will omit most of the trivial cases here, but focus on the treatment of coproducts and fixpoints.

A Patch T is an object that describe possible changes that can be made to objects of type 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 specifies how to transform a given $t_1 : T$ into a $t_2 : T$. The gapply function is performs the changes that a patch prescribes 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 : T 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 ∷ []) »= lhead
  ⋮
```

Where <$> is the applicative-style application for the *Maybe* monad; »= is the usual bind for the *Maybe* monad and lhead is the partial function of type $[a] \rightarrow Maybe\ \ a$ 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.

The case for fixpoints is handled by the gapplyL function:

```
gapplyL  : {n : ℕ}{t : T 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 »= gIns 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) »= gIns y'
```

This function proceeds by induction on the patch. In the base case, when the patch is empty, it checks that the list of values is also empty. Insertion and deletion are handled by two auxiliary functions, gIns and gDel.

Inserting a new *head* x in a list of values l is done by taking the appropriate number of recursive arguments from l, plugging x with those values and returning the result and the rest of l. This is done by the μ-close function, which uses plug internally.

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

Removing a *head* x from a a list of values l is the dual operation. We take the *head* of the first element of the list, if it matches x we then concatenate the recursive children of that first element with the rest of the list.

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

Our apply function satisfies an important correctness property. Given a *well-formed* patch p, we have that applying p to its *source* yields its *destination*:

```
gapply p (D-src p) ≡ just (D-dst p)
```

This lemma and the others relating diffing and operations over patches, provides the beginning of an equational theory of patches.

4. Residuals and Conflicts

So far, we have seen algorithms to create and apply patches, which could be used to make some simple version control system. In the real world, however, the most desired functionality of a VCS is *merging*. 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:

$$\begin{tikzcd} A_1 & A_0 \arrow[l, "p"'] \arrow[r, "q"] & A_2 \end{tikzcd}$$

Our idea, inspired by Tieleman [21], 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 1 informally illustrates the desired result of merging the patches p and q using their respective residuals:

$$\begin{tikzcd} & A_0 \arrow[dl, "p"'] \arrow[dr, "q"] & \\ A_1 \arrow[dr, "q/p"'] & & A_2 \arrow[dl, "p/q"] \\ & A_3 & \end{tikzcd}$$

Figure 1. 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.

It only makes sense to speak about the residual p/q is p and q have the same source. We say that two patches are *aligned* when they are both *well-formed* and have the same source, we denote "p is aligned with q" by $p \mid\mid q$.

It is here that the notion of *conflict* enters the stage. It is very important to clearly identify which situations we will consider as conflicts. In fact, computing a residual p/q, might give rise to the situations in figure 2.

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 [11] before deciding which changes go where. The idea is that

- 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;

Figure 2. Propagating Alice's changes, p over Bob's, q.

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 we 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. 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} → T n → U n → Set where
  UpdUpd : {n : ℕ}{t : T 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 : T n}{a : U (suc n)}
           → ValU a t → ValU a t → C t (μ a)
  UpdDel : {n : ℕ}{t : T n}{a : U (suc n)}
           → ValU a t → ValU a t → C t (μ a)
  GrowL  : {n : ℕ}{t : T n}{a : U (suc n)}
           → ValU a t → C t (μ a)
  GrowLR : {n : ℕ}{t : T n}{a : U (suc n)}
           → ValU a t → ValU a t → C t (μ a)
  GrowR  : {n : ℕ}{t : T n}{a : U (suc n)}
           → ValU a t → C t (μ a)
```

4.1 Incorporating Conflicts

Although we have now defined the data type used to represent conflicts, we still need to define our residual operator. Note that we are adding conflict information in the place of that extra parameter we discussed in Section 3.2:

```
res : {n : ℕ}{t : T n}{ty : U n}
  → (p q : Patch t ty)(hip : p || q)
  → D C t ty
```

The residual operation is defined by induction on both patches. As our patch type has quite a few constructors, the definition necessarily covers many different cases. Instead of providing the entire Agda definition here[2], we will discuss a handful of typical branches in some detail.

We begin by describing the branch when one patch changes the *head* of a fixpoint, but the other deletes it, that is, we are computing the residual:

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

We want to describe how to apply the changes $p = (\mathsf{D}\mu\text{-}\mathsf{dwn}\ dx\ :: dp)$ to a structure that has been modified by the patch $q = (\mathsf{D}\mu\text{-}\mathsf{del}\ y\ ::\ dq)$, assuming both patches have the same *source*. Well, since the destination of q has no occurrence of y at that point anymore (as it was deleted), this is going to depend on the changes dx that the patch p made to y. If dx is the identity patch, we can simply ignore it and say that $p/q = dp/dq$. If not, then we have a *update-delete* conflict at hand, so we say that $p/q = \mathsf{D}\mu\text{-}\mathsf{A}\ (\mathsf{UpdDel}\ dx\ y)\ ::\ (dp/dq)$.

The remaining cases follow a similar reasoning. For p/q the idea is to come up with a patch that can be applied to an object already modified by q but still produces the changes specified by p. When not possible we simply flag that as a conflict.

The attentive reader might have noticed a symmetric structure on our conflict data type. This is no coincidence, we can always compute the *symmetric* conflict by:

```
C-sym : {n : ℕ}{t : T n}{ty : U n}
      → C t ty → C t ty
C-sym (UpdUpd o x y) = UpdUpd o y x
C-sym (DelUpd x y) = UpdDel y x
C-sym (UpdDel x y) = DelUpd y x
C-sym (GrowL x)    = GrowR x
C-sym (GrowR x)    = GrowL x
C-sym (GrowLR x y) = GrowLR y x
```

Moreover, this symmetric structure is also present on the residual itself. Note that $\mathsf{D}\ A\ t\ ty$ is functorial on A (by construction), let D-map be its action on arrows of type $A \to B$, we can prove that for all $p, q : \mathsf{D} \perp t\ ty$, if p and q are aligned, then:

$$p/q \equiv \mathsf{D\text{-}map}\ \mathsf{C\text{-}sym}\ (\mathsf{mirror}_{p,q}(q/p))$$

Where $\mathsf{mirror}_{p,q}$ has type $\mathsf{D}\ A\ t\ ty \to \mathsf{D}\ A\ t\ ty$, for all A. This $\mathsf{mirror}_{p,q}$ will take the residual q/p and transport its structure to be that of p/q. This happens by inserting and removing $\mathsf{D}\mu\text{-}\mathsf{del}$ s where necessary.

This is a particularly interesting result, and tells us that the concepts of *residuals* and *patch commutation*, as used by Darcs [10], should not be so far apart. By carefully studying the $\mathsf{mirror}_{p,q}$ function we should be able to find sufficient conditions to prove certain *merge strategies* converge. This is the kind of result we want, in order to build a functional and reliable Version Control System.

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 3.

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

[2] The complete Agda code is publicly available and can be found in `https://github.com/VictorCMiraldo/diff-agda`.

$$\begin{tikzcd}
 & A_0 \arrow[dl, "p"'] \arrow[dr, "q"] & \\
A_1 \arrow[d, dotted, "q/p"'] & & A_2 \arrow[d, dotted, "p/q"] \\
P_{1/2} \arrow[rr, equal, "sym"] \arrow[dr, "e"'] & & P_{2/1} \arrow[dl, "e"] \\
 & A_3 &
\end{tikzcd}$$

Figure 3. Residual patch square

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 PatchC A $\rightarrow$ Patch A, that is, to turn patches containing conflicts into patches without conflicts. This type, however, would imply that e is total, therefore it could solve *all* conflicts. This is not very realistic for a lot of reasons, not to mention that some conflicts are impossible to solve automatically. To have some more freedom we shall define a *merge strategy* to be a function of type PatchC A $\rightarrow$ B (Patch A), for an arbitrary behavior monad B. For example, an interactive merge strategy would choose B to be IO; a partial merge strategy would choose B to be Maybe.

Hence the key question becomes: how to define e? As it turns out, we cannot completely answer that yet. The *merge strategy* e depends on the semantics of the files being diffed: Is the order of declarations irrelevant? Are some annotations idempotent? Where should the tool be aware of scoping? Which are the things that are *scoped*? Which sections of the file can be moved around, like a global function becoming local, freely? etc... 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: how do to traverse the PatchC structure and how to resolve the conflicts found therein.

For example, a simple pointwise, total, *merge strategy* could be defined for a function m : C t ty $\rightarrow$ D $\perp_p$ t ty, which can now be mapped over D C t ty pointwise on its conflicts. We end up with an object of type D (D $\perp_p$) t ty. Since D is a free-monad, we have a multiplication μ_D : D (D A) t ty $\rightarrow$ 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 *combinators* for describing traversals of a PatchC and how specific conflicts must be resolved. This would allow us to prove lemmas about the behavior of some *merge strategies*. More importantly, we are actively investigating how can one prove that a *merge strategy* converges.

5. Summary, Related Work and Conclusions

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 for 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 has the potential to let one define custom conflict resolution strategies, such as those that attempt to recognize the copying of subtrees. The work of Lempsink et al. [13] and Vassena [22] 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 [19] defines the antidiagonal for a type T as a type X such that there exists $X \rightarrow T^2$. That is, X produces two **distinct** T's, whereas a diff produces a T given another T.

Pijul The VCS Pijul is inspired by Mimram[14], 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 1) 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 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 was done by Jacobson [10]. 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 [4] developed such a model of patch theory.

Separation Logic Swierstra and Löh [20] 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.

5.1 Conclusion

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 and B. Reus. *Monadic Presentations of Lambda Terms Using Generalized Inductive Types*, pages 453–468. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999.

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

[4] 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.

[5] 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.

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

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

[8] 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.

[9] P. Dybjer. Inductive sets and families in martin-löf's type theory and their set-theoretic semantics. In *Logical Frameworks*, pages 280–306. Cambridge University Press, 1991.

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

[11] 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.

[12] 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.

[13] 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.

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

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

[16] P. Morris, T. Altenkirch, and C. McBride. *Exploring the Regular Tree Types*, pages 252–267. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[17] 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.

[18] 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.

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

[20] 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 &#38; Software*, Onward! '14, pages 43–54, 2014.

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

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