Structure-aware version control
A generic approach using Agda

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

Abstract
Modern version control systems are largely based on the UNIX `diff` 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 [Distributed, 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, have 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:

<table>
<thead>
<tr>
<th>Name</th>
<th>Number</th>
<th>Mark</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>440</td>
<td>7.0</td>
</tr>
<tr>
<td>Bob</td>
<td>593</td>
<td>6.5</td>
</tr>
<tr>
<td>Carroll</td>
<td>168</td>
<td>8.5</td>
</tr>
</tbody>
</table>

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 3).
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. For any type $A$ equipped with decidable equality, which indeed have the expected operations: (A) a `diff` function; and (B) an `apply` function.

$$
\text{Patch} : \text{Set} \\
\text{diff} : A \rightarrow A \rightarrow \text{Patch} \\
\text{apply} : \text{Patch} \rightarrow A \rightarrow \text{Maybe} A
$$

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.

$$
\text{hd} : \{A : \text{Set} \} \rightarrow \text{Tree} A \rightarrow \text{Maybe} A \\
\text{hd Leaf} = \text{nothing} \\
\text{hd} (\text{Node} x \_ \_) = \text{just} x
$$

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.

$$
\text{serialize} : \{A : \text{Set} \} \rightarrow \text{Tree} A \rightarrow \text{List} (\text{Tree} A) \\
\text{serialize} t = \text{hd} t :: \text{concat} (\text{map serialize} (\text{ch} t))
$$

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`.

1.2 Differing 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 differing the elements of type $A$ inside the tree.

$$
\text{data Tree} (A : \text{Set}) : \text{Set where} \\
\text{Leaf} : \text{Tree} A \\
\text{Node} : A \rightarrow \text{Tree} A \rightarrow \text{Tree} A \rightarrow \text{Tree} A
$$

The first step is to fix a $t : \text{Tree} A$ 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:

$$
\text{TreeF} : \text{Set} \rightarrow \text{Set} \\
\text{TreeF} A X = \text{Unit} \mathbin{\uplus} A \times X \times X
$$

$$
\text{Tree} : \text{Set} \rightarrow \text{Set} \\
\text{Tree} A = \text{Fix} (\text{TreeF} A)
$$

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 \cong \text{Maybe} A`.

The usual operations one expects to perform over `Tree A` are supposed to be binary trees. Before we present the data type generic definitions and algorithms, however, we will present a specific instance of our `diff` algorithm for binary trees.
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: `d1`, `d2` and `d3`. 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 to perform this transformation. We can `Del` the current `hd Leaf` and `Ins` the `hd (Node x)`, this patch would be encoded by:

\[
\text{Del nothing (Ins (just x) Nil)}
\]

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

\[
\text{Mod (diffA nothing (just x)) Nil}
\]

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. Afterwards, we will define this function for the general case later on, in Section 3.3.

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:

\[
\begin{align*}
\text{plug} : \{A : \text{Set}\} & \rightarrow (\text{as bs : List (Tree A)}) \rightarrow T \text{Patch A} \\
\text{plug} [] & = \text{Nil} \\
\text{plug} (x :: xs) & = \text{Del (hd x) (diff (ch x ++ xs) [])} \\
\text{plug} (y :: ys) & = \text{Ins (hd y) (diff [] (ch y ++ ys))} \\
\text{plug} (x :: xs) (y :: ys) & = \text{let} \\
& \quad d_1 = \text{Ins (hd y) (diff (x :: xs) (ch y ++ ys))} \\
& \quad d_2 = \text{Del (hd x) (diff (ch x ++ xs) (y :: ys))} \\
& \quad d_3 = \text{Mod (diffA (hd x) (hd y))} \\
& \quad \text{(diff (ch x ++ xs) (ch y ++ ys))} \\
& \quad \text{in } d_1 \cup d_2 \cup d_3
\end{align*}
\]

Apply 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:

\[
\begin{align*}
\text{plug} : \{A : \text{Set}\} & \rightarrow \text{Maybe A} \rightarrow \text{List (Tree A)} \\
\text{plug} & \rightarrow \text{Maybe (Tree A)} \\
\text{plug nothing} & \rightarrow \text{just Leaf} \\
\text{plug (just x) (l :: r :: ts)} & \rightarrow \text{just (Node x l r)} \\
\text{plug _ _} & \rightarrow \text{nothing}
\end{align*}
\]

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`, 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.

### 2.1 Regular Tree Types

The universe of regular tree types (sometimes also called context-free types) 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:

\[
\begin{align*}
\text{data U} & : \text{N} \rightarrow \text{Set where} \\
\text{u0} & : \{n : \text{N}\} \rightarrow \text{U n} \\
\text{u1} & : \{n : \text{N}\} \rightarrow \text{U n} \\
\text{+_+} & : \{n : \text{N}\} \rightarrow \text{U n} \rightarrow \text{U n} \rightarrow \text{U n} \\
\text{-_} & : \{n : \text{N}\} \rightarrow \text{U n} \rightarrow \text{U n} \\
\text{def} & : \{n : \text{N}\} \rightarrow \text{U (suc n)} \rightarrow \text{U n} \rightarrow \text{U n} \\
\text{var} & : \{n : \text{N}\} \rightarrow \text{U (suc n)} \\
\text{wk} & : \{n : \text{N}\} \rightarrow \text{U n} \rightarrow \text{U (suc n)}
\end{align*}
\]

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:

\[
\begin{align*}
\text{BoolU} & : \text{U 0} \\
\text{BoolU} & = \text{u1} \oplus \text{u1} \\
\text{TreeU} & : \text{U 1} \\
\text{TreeU} & = \mu (\text{u1} \oplus (\text{wk} \text{var} \oplus \text{var} \oplus \text{var})) \\
\text{BtreeU} & : \text{U 0} \\
\text{BtreeU} & = \text{def} \text{TreeU BoolU}
\end{align*}
\]

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, 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`. That is, every new variable may refer to previous variables in its definition.

\[
\begin{align*}
\text{data T} & : \text{N} \rightarrow \text{Set where} \\
[] & : \text{T 0} \\
\_ : : & : \{n : \text{N}\} \rightarrow \text{U n} \rightarrow \text{T n} \rightarrow \text{T (suc n)}
\end{align*}
\]

[1] [https://github.com/VictorMiraldo/cf-agda]
With codes and telescopes at hand, we can interpret every type expression without the need for explicit substitutions or renamings. For any code \( T \) and every telescope \( \Gamma \), we can compute a set \([T]\rceil_{\Gamma}\) as follows:

\[
\begin{align*}
[u0]\rceil_{\Gamma} &= 0 \\
[u1]\rceil_{\Gamma} &= 1 \\
[T_a \otimes T_b]\rceil_{\Gamma} &= [T_a]\rceil_{\Gamma} + [T_b]\rceil_{\Gamma} \\
[T_a \otimes T_b]\rceil_{\Gamma} &= [T_a]\rceil_{\Gamma} \times [T_b]\rceil_{\Gamma} \\
[\mu F]\rceil_{\Gamma} &= [F]_{\mu F} \\
[\var x]\rceil_{\Gamma} &= [x]_{\Gamma} \\
[\text{wk } T]\rceil_{\Gamma} &= [T]\rceil_{\Gamma} \\
[\mu T]\rceil_{\Gamma} &= [T]\rceil_{\mu T, \Gamma}
\end{align*}
\]

We will define this interpretation as an Agda datatype.

\[
data \text{EIU} : \{n : \mathbb{N}\} \rightarrow U n \rightarrow T n \rightarrow \text{Set} \where 
\begin{align*}
\text{unit} : \{n : \mathbb{N}\} \{t : T n\} &\rightarrow \text{EIU} u1 t \\
\text{inl} : \{n : \mathbb{N}\} \{t : T n\} \{a b : U n\} &\rightarrow \text{EIU} (a \otimes b) t \\
\text{inr} : \{n : \mathbb{N}\} \{t : T n\} \{a b : U n\} &\rightarrow \text{EIU} (a \otimes b) t \\
\text{top} : \{n : \mathbb{N}\} \{t : T n\} \{a : U n\} &\rightarrow \text{EIU} a t \\
\text{pop} : \{n : \mathbb{N}\} \{t : T n\} \{a : U n\} &\rightarrow \text{EIU} b t \\
\text{mu} : \{n : \mathbb{N}\} \{t : T n\} \{a : U (\text{Suc } n)\} &\rightarrow \text{EIU} a (\mu a \vdash t) \\
\text{red} : \{n : \mathbb{N}\} \{t : T n\} \{F : U (\text{Suc } n)\} \{x : U n\} &\rightarrow \text{EIU} F (x :: t) \\
\end{align*}
\]

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](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:

\[
\begin{align*}
\mu\text{-hd} : [\mu t y] t \rightarrow [t y] (u1 :: t) \\
\mu\text{-ch} : [\mu t y] t \rightarrow \text{List} ([\mu t y] t) \\
\mu\text{-plug} : [t y] (u1 :: t) \rightarrow \text{List} ([\mu t y] t) \\
\rightarrow \text{Maybe} ([\mu t y] t)
\end{align*}
\]

Moreover, \text{plug} must satisfy the expected correctness property:

\[
\forall x. \text{plug}\text{-hd} x (\text{ch} x) \equiv \text{just } x
\]

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 \( \Delta \) mapping an object \( A \) to the pair \((A, A)\) with analogous action on arrows.

In this section we will define \text{Patch}_{\Gamma} T, the type of patches over some code \( T \) and telescope \( \Gamma \). The subscripts \( \Gamma \) will be omitted when they can be inferred by the context. We will use \( \equiv \) to refer to definitions, \( \approx \) to refer to propositional equality and \( \simeq \) 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 \( T \) empty. There are no transformations one can make because there are no values to be transformed.

\[
\text{Patch } u0 = 0 \approx \Delta [u0]
\]

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

\[
\text{Patch } u1 = 1 \approx \Delta [u1]
\]

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

\[
\text{Patch } (T_a \otimes T_b) = \text{Patch } T_a \times \text{Patch } T_b
\]

\[
\approx \Delta [T_a] \times \Delta [T_b]
\]

\[
\approx \Delta [T_a \otimes T_b]
\]

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

\[
\text{Patch } (T_a \oplus T_b) = \text{Patch } T_a + \text{Patch } T_b + 2 \times [T_a] \times [T_b]
\]

\[
\approx \Delta [T_a] + 2 \times [T_a] \times [T_b] + \Delta [T_b]
\]

\[
\approx \Delta [T_a \oplus T_b]
\]

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

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

\[
\text{Patch}_{\tau, \Gamma'} \var x = \text{Patch}_{\tau, \Gamma'} \tau
\]

\[
\approx \Delta [\tau]_{\tau, \Gamma'}
\]

\[
\approx \Delta [\var x]_{\tau, \Gamma'}
\]

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

\[
\text{Patch}_{\tau, \Gamma'} (\text{wk } T) = \text{Patch}_{\tau, \Gamma'} T
\]

\[
\approx \Delta [T]_{\tau, \Gamma'}
\]

\[
\approx \Delta [\text{wk } T]_{\tau, \Gamma'}
\]

\( T \equiv \text{def } F \ x ; \) When \( T = \text{def } F \ x \), 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 \( x \).

\[
\text{Patch}^\Gamma (F \ x) = \text{Patch}_{\Gamma,F} (F)
\]

\[
\approx \Delta [F]_{\Gamma,F}
\]

\[
\approx \Delta [\text{def } F \ x]_{\Gamma}
\]

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:

\[
\text{serialize} : \{n : N\} (t : T \ n) \{ty : U \ (\text{succ } n)\}
\]

\[
\rightarrow \text{EIU } (\mu \ t) t \rightarrow \text{List } (\text{ElU } ty (u1 :: t))
\]

\[
\text{serialize } x = \mu \text{-hd } x :: \text{concat } (\text{map serialize } (\mu \text{-ch } x))
\]

This gives us a uniform way to handle fixpoint operations. 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,

\[
\text{Patch } (\mu \ F) = \text{List } (F1 + F1 + \text{Patch } (F1))
\]

This reads as “A patch of the (least) fixpoint of an \( F \)-structure is a list of edit operations over \( F1 \)”. 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 Patch \( (\mu \ F) \) is not isomorphic to \( \Delta (\mu F) \), precisely because of the partiality of deserialization.

We could define 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 \( \Sigma \)-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 \( \Sigma \)-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 \( D \ A t t y \) of diffs for the code \( t y \) 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, \( TU \rightarrow \text{Set} \), is just the type of inductive type-families over codes and telescopes, defined by \( \forall \{ n \} \rightarrow T n \rightarrow U n \rightarrow \text{Set} \).

\[
\text{data } D (A : TU \rightarrow \text{Set})
\]

\[
: \{n : N\} \rightarrow T n \rightarrow U n \rightarrow \text{Set}
\]

\[
\text{where}
\]

\[
\text{D-unit} : \{n : N\} \{t : T \ n\} \rightarrow D \ A t t u1
\]

\[
\text{D-pair} : \{n : N\} \{t : T \ n\} \{a b : U \ n\}
\]

\[
\rightarrow D \ A t a \rightarrow D \ A t b \rightarrow D \ A t (a \otimes b)
\]

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:

\[
\text{data } D (A : TU \rightarrow \text{Set})
\]

\[
: \{n : N\} \rightarrow T n \rightarrow U \ (\text{succ } n) \rightarrow \text{Set}
\]

\[
\text{where}
\]

\[
\text{D}-\text{ins} : \{n : N\} \{t : T \ n\} \{a : U \ (\text{succ } n)\}
\]

\[
\rightarrow \text{EIU } a (u1 :: t) \rightarrow D (\mu A a)
\]

\[
\text{D}-\text{del} : \{n : N\} \{t : T \ n\} \{a : U \ (\text{succ } n)\}
\]

\[
\rightarrow \text{EIU } a (u1 :: t) \rightarrow D (\mu A a)
\]

\[
\text{D}-\text{down} : \{n : N\} \{t : T \ n\} \{a : U \ (\text{succ } n)\}
\]

\[
\rightarrow D (\mu A a) \rightarrow D (\mu A a)
\]

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 Patch \( t \ ty \) as \( D (\lambda a \rightarrow \bot) t \ ty \). 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:

\[
\text{D-src} : \{A : TU \rightarrow \text{Set}\} \{n : N\} \{t : T \ n\} \{ty : U n\}
\]

\[
\rightarrow D \ A t t y \rightarrow \text{Maybe } (\text{EIU } ty)
\]

\[
\text{D-dest} : \{A : TU \rightarrow \text{Set}\} \{n : N\} \{t : T \ n\} \{ty : U n\}
\]

\[
\rightarrow D \ A t t y \rightarrow \text{Maybe } (\text{EIU } ty)
\]
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 $D_{\mu}$, 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 $\Sigma$-types, whenever necessary.

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

$$
\text{WF} : \{ A : \mathrm{TU} \to \mathrm{Set} \} \{ n : \mathbb{N} \} \{ t : T n \} \{ ty : U n \} \\
\rightarrow \mathrm{D} A t ty \rightarrow \mathrm{Set} \\
\text{WF} \{ \mathbf{\Delta} \} \{ n \} \{ t \} \{ ty \} p \\
= \Sigma (\mathrm{EIU} t y t \times \mathrm{EIU} t y t) \\
(\lambda xy \rightarrow D_{\text{src}} p \equiv \text{just } (p1 xy) \times D_{\text{dst}} p \equiv \text{just } (p2 xy))
$$

It is mechanical to prove that eliminating constructors of $D$ and $D_{\mu}$ 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 $\text{gdiff}$ that, given two elements of a regular tree type, computes the patch recording their differences. For finite types and type variables, the $\text{gdiff}$ functions follow the structure of the type in an almost trivial fashion.

$$
\text{gdiff} : \{ n : \mathbb{N} \} \{ t : T n \} \{ ty : U n \} \\
\rightarrow \text{EIU m ty t} \rightarrow \text{EIU m ty t} \rightarrow \text{Patch m t t}
$$

$$
\text{gdiff} \{ ty = \text{un} \} \text{ unit unit} \\
= D_{\text{unit}} \\
\text{gdiff} \{ ty = \text{var} \} \{ t a \} \{ t b \} \\
= D_{\text{top}} (\text{gdiff a b}) \\
\text{gdiff} \{ ty = \text{wd a} \} \{ \text{pop a} \} \{ \text{pop b} \} \\
= D_{\text{pop}} (\text{gdiff a b}) \\
\text{gdiff} \{ ty = \text{def f x} \} \{ \text{red a} \} \{ \text{red b} \} \\
= D_{\text{def}} (\text{gdiff a b}) \\
\text{gdiff} \{ ty = \text{ty ti tv} \} \{ \text{ay ay tv} \} \{ \text{by by tv} \} \\
= D_{\text{pair}} (\text{gdiff ay by}) (\text{gdiff av bv}) \\
\text{gdiff} \{ ty = \text{ty ti tv} \} \{ \text{inl a} \} \{ \text{inl b} \} \\
= D_{\text{inl}} (\text{gdiff ay by}) \\
\text{gdiff} \{ ty = \text{ty ti tv} \} \{ \text{in r a} \} \{ \text{in r b} \} \\
= D_{\text{inr}} (\text{gdiff ay by}) \\
\text{gdiff} \{ ty = \text{ty ti tv} \} \{ \text{inl a} \} \{ \text{in r b} \} \\
= D_{\text{setl ay bv}} \\
\text{gdiff} \{ ty = \text{ty ti tv} \} \{ \text{in r a} \} \{ \text{in l b} \} \\
= D_{\text{setr av by}} \\
\text{gdiff} \{ ty = m ty \} \{ a \} \{ b \} \\
= D_{\text{mu}} (\text{gdiff} a [ ] b [ ]) \\
$$

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

$$
\text{gdiffL} : \{ n : \mathbb{N} \} \{ t : T n \} \{ ty : U (\text{suc n}) \} \\
\rightarrow \text{List} (\text{ElU } m ty t) \rightarrow \text{List} (\text{ElU } m ty t) \rightarrow \text{Patch m t t ty}
$$

$$
\text{gdiffL} \{ y :: y \} [ ] = [] \\
\text{gdiffL} \{ y :: y \} [] = D_{\mu-\text{ins}} (\mu-hd y) :: (\text{gdiffL} []) (\mu-ch y ++ y) \\
\text{gdiffL} (x :: xs) [] = D_{\mu-\text{del}} (\mu-hd x) :: (\text{gdiffL} (\mu-ch x ++ xs) []) \\
\text{gdiffL} (x :: xs) (y :: yys) = \text{let} \\
\text{hdX }, \text{chX } = \mu-\text{open x} \\
\text{hdY }, \text{chY } = \mu-\text{open y} \\
d1 = D_{\mu-\text{ins}} \text{hdY } :: (\text{gdiffL} (x :: xs) (\text{chY } ++ yys)) \\
d2 = D_{\mu-\text{del}} \text{hdX } :: (\text{gdiffL} (\text{chX } ++ xs) (y :: yys)) \\
d3 = D_{\mu-\text{dwn}} (\text{gdiffL} \text{hdX hdY}) \\
= (\text{gdiffL} (\text{chX } ++ xs) (\text{chY } ++ yys))
$$

Here, $\mu-\text{open } x$ computes the pair of the head, $\mu-hd x$ and children $\mu-ch x$ 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 $\mu_{-\text{ins}}, \mu_{-\text{del}}$ 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 $\text{gdiff}$

As we mentioned previously, not all patches are well-formed. We can prove, however, that $\text{gdiff}$ is guaranteed to produce well-formed patches:

$$
D_{\text{src}} (\text{gdiff } x y) \equiv \text{just } x \\
D_{\text{dst}} (\text{gdiff } x y) \equiv \text{just } y
$$

#### Identity Patch

For all $x : [ty]_1$, we can compute the identity patch on $x$, written $D_{\text{id x}}$. Moreover, it has $x$ as its source and destination.

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

#### Inverse Patch

Given a patch $p : \text{Patch x ty}$, if it is not the identity patch, then it has some change-introduction constructors inside. We can compute the inverse patch of $p$, $D_{\text{inv}} p$ by swapping $D_{\text{setl}}$’s with $D_{\text{setr}}$’s and $D_{\mu-\text{ins}}$’s with $D_{\mu-\text{del}}$’s. It satisfies the following properties:

$$
D_{\text{src}} (D_{\text{inv}} p) \equiv D_{\text{dst}} p \\
D_{\text{dst}} (D_{\text{inv}} p) \equiv D_{\text{src}} p
$$

Therefore, if $p$ is well-formed, then $D_{\text{inv}} p$ is well-formed.

#### Composition of Patches

Given two well-formed patches $p, q : \text{Patch x ty}$, if $\text{src p} \equiv \text{dst q}$ then we can define the composition of $p$ and $q$, $p \circ q$, which also satisfies the expected properties:

$$
D_{\text{src}} (p \circ q) \equiv D_{\text{src}} q \\
D_{\text{dst}} (p \circ q) \equiv D_{\text{dst}} p
$$
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.

$$\text{cost} : \{n : \mathbb{N}\} \{t : \mathbb{T} n\} \{ty : \mathbb{U} n\} \rightarrow \text{Patch} \ t \ ty \rightarrow \mathbb{N}$$

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:

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

Remember that we call a function $\text{dist}$ a metric if the following three properties are satisfied:

1. $$\text{dist} \ x \ y = 0 \Leftrightarrow x \equiv y$$
2. $$\text{dist} \ x \ y = \text{dist} \ y \ x$$
3. $$\text{dist} \ x \ y + \text{dist} \ y \ z \geq \text{dist} \ x \ z$$

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 $\text{D-id} \ x$ should be 0, for all $x$. That is easy to achieve, as $\text{D-id} \ x$ is the patch over $x$ with no change-introduction constructor, 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 $c_{\bot} = \text{cost} \ (\text{D-} \bot \text{-del}) = \text{cost} \ (\text{D-} \bot \text{-setr})$ and $c_{\mu} = \text{cost} \ (\text{D-} \mu \text{-ins}) = \text{cost} \ (\text{D-} \mu \text{-del})$. This guarantees the second property by construction. If we define $c_{\bot}$ and $c_{\mu}$ as constants, however, the cost of inserting a small subtree will be the same cost as inserting whatever definition we gave to those values the cost function will induce a metric.

Let $\text{cost}_L = \text{sum}$, $\text{map} \ \text{cost}_\mu$, the cost function is then defined by:

$$\begin{align*}
\text{cost} \ (\text{D-A} \ ()) & = 0 \\
\text{cost} \ (\text{D-unit}) & = \text{cost} \ d \\
\text{cost} \ (\text{D-inl} \ d) & = \text{cost} \ d \\
\text{cost} \ (\text{D-inr} \ d) & = \text{cost} \ d \\
\text{cost} \ (\text{D-setl} \ xa \ xb) & = \text{cost} \ xa \ xb \\
\text{cost} \ (\text{D-setr} \ xa \ xb) & = \text{cost} \ xa \ xb \\
\text{cost} \ (\text{D-pair} \ da \ db) & = \text{cost} \ da + \text{cost} \ db \\
\text{cost} \ (\text{D-def} \ d) & = \text{cost} \ d \\
\text{cost} \ (\text{D-pop} \ d) & = \text{cost} \ d \\
\text{cost} \ (\text{D-mu} \ l) & = \text{cost}_L \ l \\
\end{align*}$$

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

$$\begin{align*}
\text{record} \ \text{Cost} : \text{Set} & \text{where} \\
\text{constructor} & \text{cost-rec} \\
\text{field} & \text{c} \bot : \{n : \mathbb{N}\} \{t : \mathbb{T} n\} \{xy : \mathbb{U} n\} \rightarrow \text{ElU} \ x t \rightarrow \text{ElU} \ y t \rightarrow \mathbb{N} \\
\text{c} \mu : \{n : \mathbb{N}\} \{t : \mathbb{T} n\} \{x : \mathbb{U} (\text{Suc} \ n)\} \rightarrow \text{ElU} \ x (\text{uU} \ t) \rightarrow \mathbb{N} \\
\text{c} \bot \text{-sym-lemma} : \{n : \mathbb{N}\} \{t : \mathbb{T} n\} \{xy : \mathbb{U} n\} \\
& \rightarrow \text{ElU} \ x (\text{uU} \ t)(\text{ey} : \text{ElU} \ y t) \\
& \rightarrow \text{c} \bot \ \text{ex ey} \equiv \text{c} \bot \text{-sym} \text{ex ey ex}
\end{align*}$$

It is straightforward to prove that the cost $(\text{D-id} \ x) \equiv 0$ and cost $(\text{D-inv} \ p) \equiv \text{cost} \ p$. For the later we need the symmetry lemma over $c_{\bot}$, 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 $c_{\bot}$ and $c_{\mu}$. 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 $c_{\mu}$ and $c_{\bot}$ 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 $c_{\bot}$ and $c_{\mu}$ in order to get the behavior we want. Nevertheless, let us take a look at where the difference between $c_{\mu}$ and $c_{\bot}$ comes into play, and calculate from there. Assume we have stopped execution of $\text{gdiffL}$ at the $d_1 \ \text{L} = d_2 \ \text{L} = d_3 \ \text{expression}$. Here we have three patches, that perform the same changes in different ways, and we have to choose one of them.

$$\begin{align*}
d_1 & = \text{D-} \mu \text{-ins} \ \text{hd} \ y : : \text{gdiffL} \ (x : : x s) (\text{ch} \ y \ + \ y s) \\
d_2 & = \text{D-} \mu \text{-del} \ \text{hd} \ x : : \text{gdiffL} \ (\text{ch} \ x \ + \ x s) (y : : y s) \\
d_3 & = \text{D-} \mu \text{-dvn} \ (\text{gdiff} \ \text{hd} \ \text{hd} \ y) : : \text{gdiffL} \ (\text{ch} \ y \ + \ x s) (\text{ch} \ y \ + \ y s)
\end{align*}$$
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 $gdiff (x :: xs) (chY + ys)$. 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:

\[
d_1 = D_{\text{\mu}} \text{-ins } hdY :: gdiff (x :: xs) (chY + ys) \\
= D_{\text{\mu}} \text{-ins } hdY :: gdiff (hdX :: chX + xs) (chY + ys) \\
= D_{\text{\mu}} \text{-ins } hdY :: D_{\text{\mu}} \text{-del } hdX \\
:: gdiff ((chX + xs) (chY + ys) \\
= D_{\text{\mu}} \text{-ins } hdY :: D_{\text{\mu}} \text{-del } hdX :: \text{tail } d_3
\]

Hence, cost $d_1$ is $c_{\mu} \cdot hdX + c_{\mu} \cdot hdY + w$, for $w = \text{cost (tail } d_3)$. Here $hdX$ and $hdY$ are values of the same type, $\text{ElU} ty$ ($\text{cons } u l t$).

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 the constructors of a (recursive) data type.

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:

\[
dist x' y' < c_{\mu} (i_j x') + c_{\mu} (i_k y') < c_{\mu} (i_j x') (i_k y')
\]

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

The choice of $c_{\mu}$ and $c_{\mu}$ 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 $\text{sizeElU}$ function and declare a top-down Cost as follows:

\[
\begin{align*}
\text{sizeElU} & : \{n : \mathbb{N}\} \{\text{t : ElU}\} \{u : \text{ElU} \rightarrow \text{ElU} u t \rightarrow \mathbb{N}\} \\
\text{sizeElU} \text{unit} & = 1 \\
\text{sizeElU} \text{inl } & = 1 + \text{sizeElU}\text{el} \\
\text{sizeElU} \text{inr } & = 1 + \text{sizeElU}\text{el} \\
\text{sizeElU} \text{ela} & = \text{sizeElU}\text{ela} + \text{sizeElU}\text{elb} \\
\text{sizeElU} \text{top el} & = \text{sizeElU}\text{el} \\
\text{sizeElU} \text{pop el} & = \text{sizeElU}\text{el} \\
\text{sizeElU} \text{red el} & = \text{sizeElU}\text{el} \\
\text{let} \text{hdE, chE} & = \mu\text{-open (mu el)} \\
\text{in } \text{sizeElU}\text{hdE} & = \text{foldr} \_ + _\_ 0 (\text{map } \text{sizeElU}\text{chE}) \\
\text{sizeElU} \text{red el} & = \text{sizeElU}\text{el}
\end{align*}
\]

\[
\begin{align*}
top-down-cost & = \text{cost-rec (\lambda ex ey } \rightarrow \text{sizeElU ex + sizeElU ey} \\
& \text{sizeElU (\lambda ex ey } \rightarrow (\_ + \text{com m (sizeElU ex) (sizeElU ey)}))
\end{align*}
\]

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 $\text{gapply}$ function is performs the changes that a patch prescribes on $t_1$, yielding $t_2$. For example, consider the case for the $D_{\text{\mu}}$ constructor, which is expecting to transform an inl $x$ into an 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:

\[
gapply : \{n: \mathbb{N}\} \{t: T n\} \{ty: U n\}
\rightarrow \text{Patch ty ty} \rightarrow \text{EU ty ty} \rightarrow \text{Maybe (EU ty ty)}
\]

\[
gapply (D\text{-}\text{inl} \text{diff}) (\text{inl el}) = \text{inl <$\text{app}$>} \text{gapply} \text{diff el}
\]

\[
gapply (D\text{-}\text{inr} \text{diff}) (\text{inr el}) = \text{inr <$\text{app}$>} \text{gapply} \text{diff el}
\]

\[
gapply (D\text{-}\text{setl} x y) (\text{inl el}) \text{ with } x = \text{U el}
\]

\[
\ldots \text{yes} \_ = \text{just (inl y)}
\]

\[
\ldots \text{no} \_ = \text{nothing}
\]

\[
gapply (D\text{-}\text{setr} y x) (\text{inr el}) \text{ with } y = \text{U el}
\]

\[
\ldots \text{yes} \_ = \text{just (inl x)}
\]

\[
\ldots \text{no} \_ = \text{nothing}
\]

\[
gapply (D\text{-}\text{setr} y x) (\text{inr el}) = \text{nothing}
\]

\[
gapply (D\text{-}\text{inl} \text{diff}) (\text{inr el}) = \text{nothing}
\]

\[
gapply (D\text{-}\text{inr} \text{diff}) (\text{inr el}) = \text{nothing}
\]

\[
gapply \{ty = \mu ty\} (D\text{-}\mu d) el = \text{gapplyL} (el :: []) \Rightarrow \text{lhead}
\]

Where $<$app> is the applicative-style application for the `Maybe` monad; $\Rightarrow$ is the usual bind for the `Maybe` monad and `lhead` is the partial function of type $[a] \rightarrow \text{Maybe} a$ 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:

\[
gapplyL : \{n: \mathbb{N}\} \{t: T n\} \{ty: U (\text{rnk n})\}
\rightarrow \text{Patch ty ty} \rightarrow \text{List (EU \_ ty ty)}
\rightarrow \text{Maybe (List (EU \_ ty ty))}
\]

\[
gapplyL [] [] = \text{just []}
\]

\[
gapplyL []_\_ = \text{nothing}
\]

\[
gapplyL (D\text{-}\text{inl} A () :: \_\_) = \text{gapplyL} (D\text{-}\text{inl} d \_ \Rightarrow \text{gapplyL} d)
\]

\[
gapplyL (D\text{-}\text{inr} A () :: \_\_) = \text{gapplyL} (D\text{-}\text{inr} d \_ \Rightarrow \text{gapplyL} d)
\]

\[
gapplyL (D\text{-}\text{setl} x \_ y :: \_\_) = \text{gapplyL} (D\text{-}\text{inr} d \_ \Rightarrow \text{gapplyL} d)
\]

\[
gapplyL (D\text{-}\text{setr} \_ y x :: \_\_) = \text{gapplyL} (D\text{-}\text{inl} d \_ \Rightarrow \text{gapplyL} d)
\]

\[
gapplyL (D\text{-}\text{inl} \_ \_ :: \_\_) = \text{gapplyL} (D\text{-}\text{inr} d \_ \Rightarrow \text{gapplyL} d)
\]

\[
gapplyL (D\text{-}\text{inr} \_ \_ :: \_\_) = \text{gapplyL} (D\text{-}\text{inl} d \_ \Rightarrow \text{gapplyL} d)
\]

\[
gapplyL t y \_ = \text{gapplyL} (D\text{-}\mu e) \text{el} = \text{gapplyL} d (el :: [] \Rightarrow \text{lhead})
\]

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 `\mu\text{-close}` function, which uses `plug` internally.

\[
gIns x l = \text{gIns} x l \Rightarrow \text{gIns} x l
\]

\[
\ldots \text{nothing} = \text{nothing}
\]

\[
\ldots \text{just (r, l') = just (r :: l')}
\]

Removing a head `x` from 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.

\[
gDel x [] = \text{nothing}
\]

\[
gDel x (y :: ys) with x = (\mu\text{-hd} y)
\]

\[
\ldots \text{True} = \text{just (\mu\text{-ch} y ++ ys)}
\]

\[
\ldots \text{False} = \text{nothing}
\]

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

\[
gapply p (D\text{-}\text{src} p) = \text{just (D\text{-}\text{dst} p)}
\]

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:

\[
A_1 \leftarrow p A_0 \rightarrow q \rightarrow A_2
\]

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 $q/p$. The diagram in Figure 1 Informally illustrates the desired result of merging the patches $p$ and $q$ using their respective residuals:

\[
A_1 \leftarrow p A_0 \rightarrow q \rightarrow A_2
\]

\[
\begin{array}{c}
A_1 \\
\downarrow p \\
A_0 \\
\uparrow q \\
A_2
\end{array}
\]

\[
\begin{array}{c}
A_3 \\
\downarrow q/p \\
A_0 \\
\uparrow p/q \\
A_2
\end{array}
\]

\[
\text{Figure 1. Residual patch square}
\]

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 $p/q$ if $p$ 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 $p \parallel q$.

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 \( a_2 \) 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 an 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:

```agda
data C : \{ n : \mathbb{N} \} \to T n \to U n \to \text{Set where}
  UpdUpd : \{ n : \mathbb{N} \} \{ t : T n \} \{ a \cdot b : U n \} \to EIU (a \uplus b) t \to EIU (a \uplus b) t \to C t (a \uplus b)
  DelUpd : \{ n : \mathbb{N} \} \{ t : T n \} \{ a : U \text{ suc n} \} \to VAU a t \to VAU a t \to C t (\mu a)
  UpdDel : \{ n : \mathbb{N} \} \{ t : T n \} \{ a : U \text{ suc n} \} \to VAU a t \to VAU a t \to C t (\mu a)
  GrowL : \{ n : \mathbb{N} \} \{ t : T n \} \{ a : U \text{ suc n} \} \to VAU a t \to C t (\mu a)
  GrowLR : \{ n : \mathbb{N} \} \{ t : T n \} \{ a : U \text{ suc n} \} \to VAU a t \to C t (\mu 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.

```agda
res : \{ n : \mathbb{N} \} \{ t : T n \} \{ ty : U n \} \to (p : \text{ Patch t ty}) (\text{ hip : p }||| q) \to 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, 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 fixedpoint, but the other deletes it, that is, we are computing the residual:

\[
(D_{dx} \triangleright y \downarrow) (D_{dy} \triangleright y \downarrow)
\]

We want to describe how to apply the changes \( p = (D_{dx} \triangleright y \downarrow) \) to a structure that has been modified by the patch \( q = (D_{dy} \triangleright y \downarrow) \), 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 \( p/q = D_{dx} \triangleright \text{UpdDel } dy \downarrow \) : \((dp/dq)\).

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 \( p \).

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:

\[
\text{C-sym} : \{ n : \mathbb{N} \} \{ t : T n \} \{ ty : U n \} \to C t ty \to C t ty
\]

\[
\text{C-sym} (\text{UpdUpd } o x y) = \text{UpdUpd } o y x
\]

\[
\text{C-sym} (\text{DelUpd } x y) = \text{DelUpd } y x
\]

\[
\text{C-sym} (\text{GrowL } x) = \text{GrowR } x
\]

\[
\text{C-sym} (\text{GrowLR } x y) = \text{GrowLR } y x
\]

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

\[
p/q \equiv D_{-} \text{C-sym} (\text{mirror }_{p,q}(q/p))
\]

Where \( \text{mirror }_{p,q} \) has type \( D \) \( t ty \to D \) \( t ty \), for all \( A \). This \( \text{mirror }_{p,q} \) will take the residual \( q/p \) and transport its structure to be that of \( p/q \). This happens by inserting and removing \( D_{\text{del-}d} \) 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 \( \text{mirror }_{p,q} \) 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.

5. Summary, Related Work and Conclusions

This is not the first paper to study the possibility of using data type generic programming for structure-aware version control. The earliest related work studies the *tree edit distance* \[7, 8, 12\]. Algorithms typically compare the Euler traversal of two trees, i.e., the string of labels encountered during a preorder traversal. The operations for transforming one tree into another is given by the list of operations transforming these Euler traversals.

In an untyped setting, there is not much to lose by flattening the tree structure. In a typed setting, however, using a list of values

\[2\] The complete Agda code is publicly available and can be found in [https://github.com/VictorCMiraldo/diff-agda](https://github.com/VictorCMiraldo/diff-agda)
to represent a patch over a tree may discard important structural information: what guarantees do we have that we can reconstruct a well-typed tree from a flattened list? It is precisely this information that we hope to preserve by adopting a data type generic approach. The work by Lempskina et al. [9] was the first to define an efficient, data type generic diff algorithm. The authors did not, however, consider the problem of merging diffs. More recently, Vassena [23] extended this work to try and define a diff3 algorithm. Both of these approaches use a heterogeneous rose tree as the underlying universe of their generic algorithms. The diff algorithm performs a linearized traversal over such rose trees.

Working with such rose trees presents several difficult problems. Patches are represented as lists of edit operations. When merging two patches, these must be aligned – that is, we need to ensure that both patches can be applied to the same trees. Vassena [23] argues that one can populate both patches with no-op edit operations, that perform no modification, in order to align them.

In this paper, we have taken a fundamentally different approach. By using a well-established universe with more structure from the outset, we hope to introduce more structure in our definition of diff data type and residual. As a result, we were hoping to avoid some of the issues with alignment and the recovery of structure that has previously been discarded that untyped algorithms face. In our experience, however, the ‘list of children’ based traversals that we have defined makes the recursive structure of our algorithms unnatural, but bearable. Reasoning with these lists of edit operations, however, becomes complex and unwieldy.

Other generic algorithms and data structures, such as zippers, generic equality, or generic parsing and pretty printing, all directly exploit the structure of the types in question, rather than flattening structure to a linear representation. We believe that this is certainly an avenue of research that is worth exploring further, even if it is not immediately clear how to do so.

Finally, there are several pieces of related work on version control systems that are worth mentioning 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 $X \rightarrow T^2$. 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 ) 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.

Conclusion
This paper tried to give a different approach to generic version control than what has been previously attempted. We have shown that even using a fundamentally different universe, we stumbled upon similar problems: modeling edits of tree-structured in a linear fashion will be problematic when one tries to merge different edits. Although we have managed to define a diff algorithm and compute with residuals, enabling us to define a diff3, reasoning about the resulting functions is not at all easy – let alone verifying the formal properties of our algorithms. We believe there is still further work to be done in this area, exploiting the inductive structure of types and trees in the merging of patches.

References


