Structure-aware version control:
A generic approach using Agda

*Victor Cacciari Miraldo*

*Wouter Swierstra*

March 2017

Department of Information and Computing Sciences
Utrecht University, Utrecht, The Netherlands
www.cs.uu.nl
Structure-aware version control
A generic approach using Agda

Victor Cacciari Miraldo Wouter Swierstra
University of Utrecht
{v.cacciari|miraldo,w.s.swierstra}@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 [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, 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 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.

$$\text{Patch} : \text{Set}$$
$$\text{Patch} \equiv A \times A$$

$$\text{diff} : A \rightarrow A \rightarrow \text{Patch}$$
$$\text{diff } x \ y = (x \cdot y)$$

$$\text{apply} : \text{Patch} \rightarrow A \rightarrow \text{Maybe } A$$
$$\text{apply } (x \cdot y) \ z \ \text{with } x == z$$
$$\mid \mid \text{True } = \text{just } y$$
$$\mid \mid \text{False } = \text{nothing}$$

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.

```haskell
data Tree (A : Set) : Set where
  Leaf : Tree A
  Node : A -> Tree A -> Tree A -> Tree A
```

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

$$\text{TreeF} : \text{Set} \rightarrow \text{Set}$$
$$\text{TreeF} A X \equiv \text{Unit } \sqcup A \times X \times X$$

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

We then deﬁne the type of the head of a Tree to be isomorphic to $\text{TreeF } A 1$, 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 $\text{TreeF } A 1 \cong \text{Maybe } A$.

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.

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

$$\text{ch} : \{ A : \text{Set} \} \rightarrow \text{Tree } A \rightarrow \text{List } (\text{TreeF } A)$$
$$\text{ch Leaf} = []$$
$$\text{ch} \ (\text{Node } _ \_ \_ r) = 1 : : r : : []$$

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.

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

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.

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 recur 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:

```haskell
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
```

With a representation of the possible transformations an element of Tree $A$ can undergo we are ready to write our ﬁrst differ 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.
\[
\text{diff} : \{A : \text{Set}\} \rightarrow (\text{as} : \text{List} (\text{Tree} A)) \rightarrow \text{Tpatch} A
\]
\[
\text{diff} \: \text{Nil} = \text{Nil}
\]
\[
\text{diff} (x :: \text{xs}) \: \text{Nil} = \text{Del} (\text{hd} x) (\text{diff} (\text{ch} x :: \text{xs}) \: \text{Nil})
\]
\[
\text{diff} (x :: \text{xs}) (y :: \text{ys}) = \text{let}
\]
\[
\quad d_1 = \text{Ins} (\text{hd} y) (\text{diff} (x :: \text{xs}) (y :: \text{ys}))
\]
\[
\quad d_2 = \text{Del} (\text{hd} x) (\text{diff} (\text{ch} x :: \text{xs}) (y :: \text{ys}))
\]
\[
\quad d_3 = \text{Mod} (\text{diff} A \: \text{nothing} (x :: \text{xs}) (y :: \text{ys}))
\]
\[
\text{in} \quad d_1 \sqcup d_2 \sqcup d_3
\]

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 \text{Maybe} A, but now, additionally, we can also compare the \text{Maybe} A values on the beginning of both lists and try to change one into the other. This is done by the \text{diff} A function. Afterwards, we have to choose one of the three patches we have: \(d_1, d_2\), or \(d_3\). The associative operator \(\sqcup\) simply chooses the patch with the least cost.

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

\[
\text{Del} \text{nothing} (\text{Ins} (\text{just} x) \text{Nil}) \quad \text{(p.1)}
\]

Or, we could \text{Mod} the constructor from a \text{Leaf} into a \text{Node} \(x\):

\[
\text{Mod} (\text{diff} A \text{nothing} (\text{just} x)) \text{Nil} \quad \text{(p.2)}
\]

The \text{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 \text{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 \text{Node}. That is, patch p.1 describes the actual changes better than patch p.2. Hence, patch p.1 should have a lower \text{cost}.

When we say we want patches to be minimal, we are referring to them having a minimal \text{cost}. Thus, the \text{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 \text{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 \text{plug} function to reassemble trees from their head and children. In this example, we can define the \text{plug} function as follows:

\[
\text{plug} : \{A : \text{Set}\} \rightarrow \text{Maybe} A \rightarrow \text{List} (\text{Tree} A)
\]

\[
\text{plug} \text{nothing} = \text{just} \text{Leaf}
\]
\[
\text{plug} (\text{just} x) (l :: r :: ts) = \text{just} (\text{Node} x l r)
\]
\[
\text{plug} = \text{nothing}
\]

The \text{apply} function has to be partial, for the same reason that \text{plug} is partial: if we are \text{plugging} a \text{just}, we need at least two \text{Trees}. This is not a problem as we can prove that the \text{patches} produced and manipulated by our algorithms are \text{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 \text{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 \text{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.

### 2.1 Regular Tree Types

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

The type of \text{codes} with \(n\) (de Bruijn style) type variables is defined by:

\[
data U : \text{N} \rightarrow \text{Set} \text{where}
\]
\[
\quad u_0 : \{n : \text{N}\} \rightarrow \text{U} n
\]
\[
\quad u_1 : \{n : \text{N}\} \rightarrow \text{U} n
\]
\[
\quad \oplus : \{n : \text{N}\} \rightarrow \text{U} n \rightarrow \text{U} n \rightarrow \text{U} n
\]
\[
\quad \otimes : \{n : \text{N}\} \rightarrow \text{U} n \rightarrow \text{U} n \rightarrow \text{U} n
\]
\[
\quad \text{def} : \{n : \text{N}\} \rightarrow \text{U} (\text{Suc} n) \rightarrow \text{U} n \rightarrow \text{U} n
\]
\[
\quad \mu : \{n : \text{N}\} \rightarrow \text{U} (\text{Suc} n) \rightarrow \text{U} n
\]
\[
\quad \text{var} : \{n : \text{N}\} \rightarrow \text{U} (\text{Suc} n)
\]
\[
\quad \text{wk} : \{n : \text{N}\} \rightarrow \text{U} n \rightarrow \text{U} (\text{Suc} n)
\]

The \text{N} index gives the number of free type variables available in the expression. The most recently bound variable may be referred to using the \text{var} constructor; the weakening constructor \text{wk} discards the topmost variable, allowing access to the others. The least fixpoint, \(
\mu\), and definitions, \text{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:

\[
\begin{align*}
\text{boolU} : \text{U} 0 \\
\text{treeU} : \text{U} 1 \\
\text{btreeU} : \text{U} 0
\end{align*}
\]

\[
\text{btreeU} = \text{def} \text{treeU} \text{boolU}
\]

Here we use the \text{def} constructor to instantiate the \text{treeU} type.

We now need to provide an interpretation function that maps a given code, in \text{U}, to a \text{Set}. On a first try, it would be natural to attempt interpreting only \text{closed} type expressions, \text{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 \text{telescope} [9]. That is, every new variable may refer to previous variables in its definition.

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

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 $\Gamma$, we can compute a set $[T]_{\Gamma}$ as follows:

\[
\begin{align*}
[u0]_{\Gamma} &= 0 \\
[u1]_{\Gamma} &= 1 \\
(T + T)_{\Gamma} &= [T]_{\Gamma} + [T]_{\Gamma} \\
(T \times T)_{\Gamma} &= [T]_{\Gamma} \times [T]_{\Gamma} \\
[\text{def } F x]_{\Gamma} &= [F]_{x,\Gamma} \\
[\text{var } x]_{\Gamma} &= [x]_{\Gamma} \\
[\text{wk } T]_{x,\Gamma} &= [T]_{\Gamma} \\
[\mu T]_{\Gamma} &= [T]_{\mu,\Gamma}
\end{align*}
\]

We will define this interpretation as an Agda datatype.

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

\[
\begin{align*}
\text{unit} &: \{n : N\} \{t : T\} n \rightarrow \text{EIU}_{u1} t \\
\text{inl} &: \{n : N\} \{t : T\} \{a,b : U\} n \rightarrow \text{EIU}_{\text{inl}} (a \oplus b) t \\
\text{inr} &: \{n : N\} \{t : T\} \{a,b : U\} n \rightarrow \text{EIU}_{\text{inr}} (a \oplus b) t \\
\\cdot\cdot\cdot \\
\text{top} &: \{n : N\} \{t : T\} a : U n \rightarrow \text{EIU}_{\text{top}} a t \\
\text{pop} &: \{n : N\} \{t : T\} \{a,b : U\} n \rightarrow \text{EIU}_{\text{pop}} b t \\
\text{mu} &: \{n : N\} \{t : T\} \{a : U\} (\text{succ} n) n \rightarrow \text{EIU}_{\text{mu}} a t \\
\text{red} &: \{n : N\} \{t : T\} \{x : U\} (\text{def } x) n \rightarrow \text{EIU}_{\text{red}} (\text{def } x) t
\end{align*}
\]

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 [GitHub](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:

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

Moreover, 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 $\text{Patch}_T$, 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, $\cong$ 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.

\[
T \equiv \text{a0} ; \quad \text{When } T \text{ is the empty type, the type of patches is on } T \text{ empty. There are no transformations one can make because there are no values to be transformed.}
\]

\[
\text{Patch } a0 = 0 \equiv \Delta[a0]
\]

\[
T \equiv \text{a1} ; \quad \text{When } T \text{ is the unit type, there is only one possible transformation: no change at all.}
\]

\[
\text{Patch } a1 = 1 \equiv \Delta[a1]
\]

\[
T \equiv T_a \otimes T_b ; \quad \text{When } T \text{ is a product of two types, again, there is only one possible transformation: to transform the components of the pair separately:}
\]

\[
\begin{align*}
\text{Patch } (T_a \otimes T_b) &= \text{Patch } T_a \times \text{Patch } T_b \\
&\approx \Delta[T_a] \times \Delta[T_b] \\
&\approx \Delta[T_a \otimes T_b]
\end{align*}
\]

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

\[
\begin{align*}
\text{Patch } (T_a \oplus T_b) &= \text{Patch } T_a + \text{Patch } T_b + 2 \times [T_a] \times [T_b] \\
&\approx \Delta[T_a] + 2 \times [T_b] + \Delta[T_b] \\
&\approx \Delta[T_a \oplus T_b]
\end{align*}
\]

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 $\text{var}$, $\text{wk}$ and $\text{def}$ 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.

\[
T \equiv \text{var} ; \quad \text{When } T \text{ is the topmost variable, we can assert that we have at least one element on } \Gamma, \text{ hence } \Gamma = \Gamma'.
\]

\[
\text{Patch } \tau \text{var } \equiv \text{Patch } \tau \cong \Delta[\tau] \cong \Delta[\text{var }] \cong \Delta[\tau]' \cong \Delta[\tau']
\]

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

\[
\text{Patch } \text{wk } T = \text{Patch } T \equiv \Delta[T] \cong \Delta[\text{wk } T] \cong \Delta[\text{wk } T]' \cong \Delta[\text{wk } T']
\]

\[
T \equiv \text{def } F x ; \quad \text{When } T = \text{def } F x, \text{ we simply need to patch } F, \text{ adding } x \text{ to the telescope in order to bind the topmost variable,}
\]

\[
\text{Patch } \text{def } F x = \Delta[F] \text{var } x = \Delta[F] \cong \Delta[\text{def } F x] \cong \Delta[\text{def } F x]'.
\]
that is, de Bruijn index 0, of $F$ to $x$.

$$\text{Patch}^\Gamma (F \ x) = \text{Patch}^{\varepsilon, \Gamma} F \approx \Delta [F]_{\varepsilon, \Gamma} \approx \Delta [\text{def } F] x, \Gamma$$

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:

$$\text{serialize} : \{n : N\} \{t : T\ n\} \{ty : U (\text{suc } n)\}$$
$$\rightarrow \text{EIU} (\mu ty t \rightarrow \text{List} (\text{EIU ty } \text{a1 : t}))$$

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,

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

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

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

$$\text{data D} (A : \text{TU} \rightarrow \text{Set}) : \{n : N\} \rightarrow T \ n \rightarrow U \ n \rightarrow \text{Set}$$

where

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

$$\text{D-pair} : \{n : N\} \{t : T \ n\} \{a b : U \ n\}$$
$$\rightarrow D \ A \ t \ a \rightarrow D \ A \ t \ b \rightarrow D \ A \ t \ (a \odot b)$$

Besides the definitions for the basic type constructors, as we presented previously, the $D$ 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:

$$\text{data D}_\mu (A : \text{TU} \rightarrow \text{Set}) : \{n : N\} \rightarrow T \ n \rightarrow (A \ 1 + A \ n) \rightarrow \text{Set}$$

where

$$\text{D}_{\mu} - \text{ins} : \{n : N\} \{t : T \ n\} \{a : U (\text{suc } n)\}$$
$$\rightarrow \text{EIU } a (\text{u1 : t}) \rightarrow D\_{\mu} A \ t \ a$$

$$\text{D}_{\mu} - \text{del} : \{n : N\} \{t : T \ n\} \{a : U (\text{suc } n)\}$$
$$\rightarrow \text{EIU } a (\text{u1 : t}) \rightarrow D\_{\mu} A \ t \ a$$

$$\text{D}_{\mu} - \text{down} : \{n : N\} \{t : T \ n\} \{a : U (\text{suc } n)\}$$
$$\rightarrow D \ A (\text{u1 : t}) \ a \rightarrow D\_{\mu} A \ t \ a$$

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 $\text{Patch} t \ ty$ as $D (\lambda_{\_ \rightarrow \bot} \ t \ ty)$. In other words, a $\text{Patch}$ is a $D$ structure that never uses the $D$ 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:

$$\text{D-src} : \{A : \text{TU} \rightarrow \text{Set}\} \{n : N\} \{t : T \ n\} \{ty : U \ n\}$$
$$\rightarrow D \ A \ t \ ty \rightarrow \text{Maybe} (\text{EIU ty})$$

$$\text{D-dst} : \{A : \text{TU} \rightarrow \text{Set}\} \{n : N\} \{t : T \ n\} \{ty : U \ n\}$$
$$\rightarrow D \ A \ t \ ty \rightarrow \text{Maybe} (\text{EIU ty})$$
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 \( D \), 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:

\[
\text{WF} : \{ A : \text{U} \rightarrow \text{Set} \} \rightarrow \{ n : \mathbb{N} \} \{ t : T n \} \{ ty : U n \} \rightarrow D A t ty \rightarrow \text{Set}\\
\text{WF } \{ A \} \{ n \} \{ \{ ty \} p \} =\Sigma (\text{EU} ty t \times \text{EU} ty t)\\
(\lambda x y \rightarrow D\text{-src } p \equiv \text{just } (p1 x y) \times D\text{-dst } p \equiv \text{just } (p2 x y))
\]

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 follows the structure of the type in an almost trivial fashion.

\[
\text{gdiff} : \{ n : \mathbb{N} \} \{ t : T n \} \{ ty : U n \} \rightarrow \text{EU ty t} \rightarrow \text{EU ty t} \\
\rightarrow \text{Patch } t ty\\
\text{gdiff } \{ ty = u1 \} = \text{unit } unit\\
\text{gdiff } \{ ty = var \} = D\text{-unit}\\
\text{gdiff } \{ ty = wk a \} = D\text{-top } \{ \text{gdiff } a b \}\\
\text{gdiff } \{ ty = def x \} = D\text{-pop } \{ \text{gdiff } a b \}\\
\text{gdiff } \{ ty = tv \} = D\text{-def } \{ \text{gdiff } a b \}\\
\text{gdiff } \{ ty = tv \} = D\text{-pair } \{ \text{gdiff } ay by \}\\
\text{gdiff } \{ ty = tv \} = D\text{-inl } \{ \text{gdiff } ay by \}\\
\text{gdiff } \{ ty = tv \} = D\text{-inr } \{ \text{gdiff } ay by \}\\
\text{gdiff } \{ ty = tv \} = D\text{-setl } ay by\\
\text{gdiff } \{ ty = tv \} = D\text{-setr } ay by\\
\text{gdiff } \{ ty = \mu \} = D\text{-mu } \{ \text{gdiff } \{ a :: [] \} \{ b :: [] \} \}
\]

Diffing 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:

\[
\text{gdiffL} : \{ n : \mathbb{N} \} \{ t : T n \} \{ ty : U (\text{suc } n) \} \\
\rightarrow \text{List } (\text{EU } \{ \mu \} t) t \rightarrow \text{List } (\text{EU } \{ \mu \} t) t \\
\rightarrow \text{Patch } t ty\\
\text{gdiffL } \{ y : \mathbb{Y} \} = [ ]\\
\text{gdiffL } \{ y :: \mathbb{Y} \} = \text{D}\text{-ins } \{ \text{mu-hd } y \} (\{ \text{gdiffL } \{ y \} \} (\text{mu-ch } y + + y))\\
\text{gdiffL } \{ x :: xs \} = \text{D}\text{-del } \{ \text{mu-hd } x \} (\{ \text{gdiffL } \{ y : \mathbb{Y} \} \} (\text{mu-ch } x + + xs))\\
\text{gdiffL } \{ x :: xs \} (y :: \mathbb{Y}) = \text{let }\\
hdX , \text{chX } = \mu\text{-open } x\\
hdY , \text{chY } = \mu\text{-open } y\\
d1 = \text{D}\text{-ins } \text{hdY } : (\{ \text{gdiffL } \{ x :: xs \} (chY + + ys) \})\\
d2 = \text{D}\text{-del } \text{hdX } : (\{ \text{gdiffL } (chX + + xs) (y :: \mathbb{Y}) \})\\
d3 = \text{D}\text{-dwn } (\{ \text{gdiffL } (chX + + xs) (chY + + ys) \})\\
in d1 d2 d3
\]

Here, \( \mu\text{-open } x \) computes the pair of the head, \( \mu\text{-hd } x \) and children \( \mu\text{-ch } x \) 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 \( y : \mathbb{Y} \), we need to insert the respective head and add the children to the forest; and finally, to transform \( x : \mathbb{X} \) into \( [ ] \) we need to delete the respective values. The interesting case happens when we want to transform \( x : \mathbb{X} \) into \( y : \mathbb{Y} \). Here we have three possible diffs that perform the required transformation. We want to choose the diff with the least cost. The associative operator \( \_ \odot \_ \_ \_ \) 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:

\[
\text{D}\text{-src } (\text{gdiff } \mu \_ x y) \equiv \text{just } x\\
\text{D}\text{-dst } (\text{gdiff } \mu \_ x y) \equiv \text{just } y
\]

#### Identity Patch
For all \( x : \{ \text{ty} \} r \), we can compute the identity patch on \( x \), written \( \text{D-Id } x \). 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 \( p : \text{Patch } t ty \), 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:

\[
\text{D}\text{-src } (\text{D}\text{-inv } p) \equiv \text{D}\text{-dst } p\\
\text{D}\text{-dst } (\text{D}\text{-inv } p) \equiv \text{D}\text{-src } p
\]

Therefore, if \( p \) is well-formed, then \( \text{D}\text{-inv } p \) is well-formed.

#### Composition of Patches
Given two well-formed patches \( p, q : \text{Patch } t ty \), if \( \text{D}\text{-src } p \equiv \text{D}\text{-dst } q \) then we can define the composition of \( p \) and \( q \), \( p \odot q \), which also satisfies the expected properties:

\[
\text{D}\text{-src } (p \odot q) \equiv \text{D}\text{-src } q\\
\text{D}\text{-dst } (p \odot q) \equiv \text{D}\text{-dst } p
\]
### 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.

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

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:

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

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

\[
\begin{align*}
\text{dist} \ x \ y &= 0 \iff x = y \\
\text{dist} \ x \ y &= \text{dist} \ y \ x \\
\text{dist} \ x \ y + \text{dist} \ y \ z &\geq \text{dist} \ x \ z
\end{align*}
\]

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

Equation (1) tells us 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 \text{change-introduction} constructors, we just assign a cost of 0 to every non-\text{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 \text{change-introduction} constructors to their dual counterpart. We will hence assign a cost \( c_\oplus = \text{cost} \ (\text{D-setl} \ x \ y) = \text{cost} \ (\text{D-setr} \ x \ y) \) and \( c_\mu = \text{cost} \ (\text{D-ins} \ x \ y) = \text{cost} \ (\text{D-del} \ x \ y) \). This guarantees the second property by construction. If we define \( c_\oplus \) and \( c_\mu \) 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 \( c_\mu \) will be functions, \( c_\oplus \ x \ y = \text{cost} \ (\text{D-setl} \ x \ y) = \text{cost} \ (\text{D-setr} \ x \ y) \) and \( c_\mu \ x = \text{cost} \ (\text{D-ins} \ x \ y) = \text{cost} \ (\text{D-del} \ x \ y) \). 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 \text{change-introduction} constructors in a patch \( p \) by \( \#p \). In the best case scenario, \( \#(\text{gdiff} \ x \ y) + \#(\text{gdiff} \ y \ z) > \#(\text{gdiff} \ x \ z) \), 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 \( \#(\text{gdiff} \ x \ y) + \#(\text{gdiff} \ y \ z) > \#(\text{gdiff} \ x \ z) \), and since the \text{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 \( \text{costL} = \text{sum} \cdot \text{map} \ (\text{cost} \mu) \), the \text{cost} function is then defined by:

\[
\begin{align*}
\text{cost} \ (\text{D-A} \ ()) &= 0 \\
\text{cost} \ (\text{D-unit} \ x) &= \text{cost} \ x \\
\text{cost} \ (\text{D-inl} \ d) &= \text{cost} \ d \\
\text{cost} \ (\text{D-inr} \ d) &= \text{cost} \ d \\
\text{cost} \ (\text{D-setl} \ xa \ xb) &= c_\oplus \ xa \ xb \\
\text{cost} \ (\text{D-setr} \ xa \ xb) &= c_\oplus \ xa \ xb \\
\text{cost} \ (\text{D-pair} \ da \ db) &= \text{cost} \ da + \text{cost} \ db \\
\text{cost} \ (\text{D-def} \ d) &= \text{cost} \ d \\
\text{cost} \ (\text{D-top} \ d) &= \text{cost} \ d \\
\text{cost} \ (\text{D-pop} \ d) &= \text{cost} \ d \\
\text{cost} \ (\text{D-mul} \ l) &= \text{costL} \ l
\end{align*}
\]

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

\[
\text{record Cost} : \text{Set where}
\begin{align*}
\quad &\text{constructor cost-rec} \\
\quad &\quad \text{field} \\
\quad &\quad \quad c_\oplus : \{ n : \mathbb{N} \}\{ t : T n \}\{ x y : U n \} \\
\quad &\quad \quad \rightarrow \text{EIU} \ x t \rightarrow \text{EIU} \ y t \rightarrow \mathbb{N} \\
\quad &\quad \quad c_\mu : \{ n : \mathbb{N} \}\{ t : T n \}\{ x : U (\text{succ} \ n) \} \\
\quad &\quad \quad \rightarrow \text{EIU} \ x \ (u1 :: t) \rightarrow \mathbb{N}
\end{align*}
\]

\[
\text{c-\oplus-l} : \{ n : \mathbb{N} \}\{ t : T n \}\{ x y : U n \} \\
\quad \rightarrow (\text{ex} : \text{EIU} \ x t) (\text{ey} : \text{EIU} \ y t) \\
\quad \rightarrow c_\oplus \ ex \ ey \equiv c_\oplus \ ey \ ex
\]

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

\[
\begin{align*}
\text{d}_1 &= \text{D-ins} \ hdY :: \text{gdiffL} \ (x :: xs) \ (chY + ys) \\
\text{d}_2 &= \text{D-del} \ hdX :: \text{gdiffL} \ (chX + xs) \ (y :: ys) \\
\text{d}_3 &= \text{D-dvn} \ (\text{gdiffL} \ hdX hdY) \\
&:: \text{gdiffL} \ (chX + xs) \ (chY + ys)
\end{align*}
\]
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 \( gdiffL (x :: xs) (chY + ys) \). 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:

\[
\begin{align*}
d_1 &= D_{\mu}\text{-ins } hdY :: gdiffL (x :: xs) (chY + ys) \\
     &= D_{\mu}\text{-ins } hdY :: gdiffL (hdX :: chX + xs) (chY + ys) \\
     &= D_{\mu}\text{-ins } hdY :: D_{\mu}\text{-del } hdX \\
     &\quad :: gdiffL (chX + xs) (chY + ys) \\
     &= D_{\mu}\text{-ins } hdY :: D_{\mu}\text{-del } hdX :: \text{tail } d_3
\end{align*}
\]

Hence, \( d_1 \) is \( c_\mu \) \( hdX + c_\mu \) \( hdY + w \), for \( w = \text{cost} (\text{tail } d_3) \). Here \( hdX \) and \( hdY \) are values of the same type, \( \text{EIU} ty (\text{toons } u1 i) \).

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 \( hdX \) and \( hdY \). If \( hdX \) becomes \( D_{\text{setl}} x' y' \) or a \( D_{\text{setr}} y' x' \), hence the cost of \( d_3 \) becomes \( c_{\text{el}} x' y' + w \). 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 for the \( c_{\mu} \) and \( c_{\text{el}} \) needs to satisfy in order to obtain the desired behavior.

\[
\begin{align*}
\text{cost } d_1 &< \text{cost } d_3 \\
\iff c_\mu (i_j x') + c_\mu (i_k y') + w &< c_{\text{el}} (i_j x') (i_k y') + w \\
\iff c_\mu (i_j x') + c_\mu (i_k y') &< c_{\text{el}} (i_j x') (i_k y')
\end{align*}
\]

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 preserve the constructor information. We now have \( hdX = i_j x' \) and \( hdY = i_j y' \), the cost of \( d_3 \) still is \( c_\mu (i_j x') + c_\mu (i_k y') + w \) but the cost of \( d_3 \) will be \( \text{cost } (gdiff (i_j x') (i_j y')) + w \). Since \( gdiff (i_j x') (i_j y') \) will reduce to \( gdiff x' y' \) preceded by a sequence of \( D_{\text{inr}} \) and \( D_{\text{inr}} \), which have zero cost. Hence, \( \text{cost } d_3 = \text{cost } (gdiff x' y') + w \).

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 \( \text{cost} \) function will need to respect:

\[
\begin{align*}
\text{cost } d_3 &< \text{cost } d_1 \\
\iff \text{dist } x' y' + w &< c_\mu (i_j x') + c_\mu (i_j y') + w \\
\iff \text{dist } x' y' &< c_\mu (i_j x') + c_\mu (i_j y')
\end{align*}
\]

Recall that our objective was to calculate a specification for the \( \text{cost} \) 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:

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

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 \( c_{\text{el}} \) that encourages the diff algorithm to favor (smaller) changes further down in the tree.

The choice of \( c_\mu \) and \( c_{\text{el}} \) 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 \( \text{Cost} \) 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 \( \text{sizeElEU} \) function and declare a top-down \( \text{Cost} \) as follows:

\[
\begin{align*}
\text{sizeElEU} : \{ n : N \} \{ t : T \} \{ w : U \} \rightarrow \text{EIU} u t \rightarrow N \\
\text{sizeElEU } \text{unit} &= 1 \\
\text{sizeElEU } (\text{inl } el) &= 1 + \text{sizeElEU } el \\
\text{sizeElEU } (\text{inr } el) &= 1 + \text{sizeElEU } el \\
\text{sizeElEU } (\text{ela } . . \text{elb}) &= \text{sizeElEU } \text{ela} + \text{sizeElEU } \text{elb} \\
\text{sizeElEU } (\text{top } el) &= \text{sizeElEU } el \\
\text{sizeElEU } (\text{pop } el) &= \text{sizeElEU } el \\
\text{sizeElEU } (\text{mu } el) &= \text{sizeElEU } \text{red } el \\
&= \text{top-down-cost } \lambda \text{ex ey} \rightarrow \text{sizeElEU } \text{ex} + \text{sizeElEU } \text{ey} \\
&\text{sizeElEU } (\lambda \text{ex ey} \rightarrow (+\text{com m} \text{sizeElEU } \text{ex} \text{sizeElEU } \text{ey}))
\end{align*}
\]

### 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 \( \text{inl } x \) to \( \text{inl } x' \). What should be the result of applying \( p \) to the value \( \text{inr } y \)? 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 \( \text{Maybe } T \).

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 \( D_{\text{setl}} \) constructor, which is expecting to transform an \( \text{inl } x \) into a \( \text{inr } y \). Upon receiving a \( \text{inl } \) value, we need to check whether or not its contents are equal to \( x \). If this holds, we can simply return \( \text{inr } y \) as intended. If not, we fail and return \( \text{nothing} \).
The definition of the \texttt{gapply} function proceeds by induction on the patch:

\[
\text{gapply} : \{ n : \mathbb{N} \} \{ t : T_n \} \{ ty : U_n \} \\
\rightarrow \text{Patch} \ t \ ty \rightarrow \text{EIU} \ ty \ t \rightarrow \text{Maybe} \ (\text{EIU} \ ty \ t)
\]

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

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

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

... \text{yes} = \text{just} \ (\text{inr} \ y)

... \text{no} = \text{nothing}

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

... \text{yes} = \text{just} \ (\text{inl} \ x)

... \text{no} = \text{nothing}

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

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

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

Where \<$>\>\> is the applicative-style application for the Maybe monad; \(=\>\>\>\> is the usual bind for the Maybe monad and \llq lhead\rrq 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 \text{gapply} for coproducts is reasonably straightforward.

The case for fixpoints is handled by the \text{gapplyL} function:

\[
\text{gapplyL} : \{ n : \mathbb{N} \} \{ t : T_n \} \{ ty : U (\text{suc} \ n) \} \\
\rightarrow \text{Patch} \ \mu \ t \ ty \rightarrow \text{List} \ (\text{EIU} \ (\mu \ ty) \ t) \\
\rightarrow \text{Maybe} \ (\text{List} \ (\text{EIU} \ (\mu \ ty) \ t))
\]

\[
\text{gapplyL} \ (\llq \ >\llq ) = \text{just} \ []
\]

\[
\text{gapplyL} \ (\llq \ >\llq ) = \text{nothing}
\]

\[
\text{gapplyL} \ (D\text{-mu} \ A \ (\llq \ >\llq ))
\]

\[
\text{gapplyL} \ (D\text{-mu} \ x :: d) \ l = \text{gapplyL} \ d \ l \Rightarrow \text{glns} \ x
\]

\[
\text{gapplyL} \ (D\text{-mu} \ x :: d) \ l = \text{gDel} \ x \ l \Rightarrow \text{gapplyL} \ d
\]

\[
\text{gapplyL} \ (D\text{-mu} \ d :: d) \ (\llq \ >\llq ) = \text{nothing}
\]

\[
\text{gapplyL} \ (D\text{-mu} \ d :: d) \ (y :: l) \ \text{with} \ \mu\text{-open} \ y
\]

... \text{hdY} , \text{chY} \text{with} \text{gapply} \ \text{dx} \ \text{hdY}

... \text{nothing} = \text{nothing}

... \text{just} \ y = \text{gapplyL} \ d \ (\text{chY} \ y :: l) \Rightarrow \text{glns} \ 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, \text{glns} and \text{gDel}.

Inserting a new \text{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 \text{plug} internally.

\[
\text{glns} \ x \ l \ \text{with} \ \mu\text{-close} \ x \ l
\]

... \text{nothing} = \text{nothing}

... \text{just} \ (r , l') = \text{just} \ (r : l')

Removing a \text{head} \( x \) from a a list of values \( l \) is the dual operation. We take the \text{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.
• 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 \(\text{Patches}\) 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:

\[
\text{data C : } \{n : \mathbb{N}\} \to T n \to U n \to \text{Set where}
\]

\[
\text{UpdUpd} : \{n : \mathbb{N}\} \{t : T n\} \{a : U n\} \to EIU (a \oplus b) t \to EIU (a \oplus b) t \to EIU (a \oplus b) t \to C t (a \oplus b)
\]

\[
\text{DelUp} : \{n : \mathbb{N}\} \{t : T n\} \{a : U \text{suc } n\} \to \text{ValU } a t \to \text{ValU } a t \to C t (\mu a)
\]

\[
\text{UpdDel} : \{n : \mathbb{N}\} \{t : T n\} \{a : U \text{suc } n\} \to \text{ValU } a t \to \text{ValU } a t \to C t (\mu a)
\]

\[
\text{GrowL} : \{n : \mathbb{N}\} \{t : T n\} \{a : U \text{suc } n\} \to \text{ValU } a t \to C t (\mu a)
\]

\[
\text{GrowLR} : \{n : \mathbb{N}\} \{t : T n\} \{a : U \text{suc } n\} \to \text{ValU } a t \to C t (\mu a)
\]

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:

\[
\text{res} : \{n : \mathbb{N}\} \{t : T n\} \{y : U n\} \to \text{p q } \text{Patch ty}(\text{hip } p || q) \to D C t ty
\]

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\(^2\), 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 fixed-point, but the other deletes it, that is, we are computing the residual:

\[
(\text{D}_{\mu \text{-dwn}} dx : : dp)/(\text{D}_{\mu \text{-del}} y : : dq)
\]

We want to describe how to apply the changes \(p = (\text{D}_{\mu \text{-dwn}} dx : : dp)\) to a structure that has been modified by the patch \(q = (\text{D}_{\mu \text{-del}} y : : dq)\), 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 an update-delete conflict at hand, so we say that \(p/q = \text{D}_{\mu \text{-A}} (\text{UpdDel } dx y) :: (dp/dq)\).

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:

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

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

\[
\text{C-sym} (\text{DelUp } dx y) = \text{DelUp } dy x
\]

\[
\text{C-sym} (\text{UpdDel } dx y) = \text{DelUp } dy x
\]

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

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

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

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

Where \(\text{mirror}_{p,q}\) has type \(D \downarrow A \times t \to D \downarrow A \times t\), 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 \(\text{D}_{\mu \text{-del}}\) as 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 [GitHub](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 Lempsink et al. [13] 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 \( X \) such that there exists \( X \to T^2 \). That is, \( X \) produces two `distinct` \( T \)'s, whereas a diff produces a \( T \) given another \( T \).

**Pijul** The VCS Pijul is inspired by 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.

**Darc** The canonical example of a formal VCS is Darc [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**


