Type-Directed Diffing of Structured Data

Victor Cacciari Miraldo  
Computing Sciences  
University of Utrecht  
v.cacciarimiraldo@uu.nl

Pierre-Évariste Dagand  
Laboratoire d’Informatique de Paris 6  
pierre-eariste.dagand@lip6.fr

Wouter Swierstra  
Computing Sciences  
University of Utrecht  
w.s.swierstra@uu.nl

Abstract
The Unix `diff` utility that compares lines of text is used pervasively by version control systems. Yet certain changes to a program may be difficult to describe accurately in terms of modifications to individual lines of code. As a result, observing changes at such a fixed granularity may lead to unnecessary conflicts between different edits. This paper presents a generic representation for describing transformations between algebraic data types and a non-deterministic algorithm for computing such representations. These representations can be used to give a more accurate account of modifications made to algebraic data structures – and the abstract syntax trees of computer programs in particular – as opposed to only considering modifications between their textual representations.

CCS Concepts  
-Software and its engineering → Functional languages; Data types and structures; -Applied computing → Version control;

Keywords  
Datatype Generic Programming, Version Control, Dependently-typed programming, Agda

1 Introduction
Programming has become a collaborative activity. Besides external contributors, we collaborate first and foremost with our future selves. Most software projects nowadays adopt a version control system to record their history, track changes and enable the concurrent exploration of various lines of thoughts. Both activities – programming and tracking change – are tightly coupled, as witnessed by various curricula (Cochez et al. 2013; MacWilliam 2013) that teach both programming and version control in a holistic manner. Indeed, a disciplined programmer will adopt a programming style that yields a readable history of changes by, for example, applying small and incremental modifications forming a coherent whole.

This methodology lies at the heart of the development of the Linux kernel, for which a single release involves around 4000 developers concurrently modifying the code at a rate of about 9 changes per hour (Kroah-Hartman 2016). Similar tools and techniques have been applied in academic circles, a striking example being the HoTT book (Univalent Foundations Program 2013) written over the course of a few months by about 20 contributors (Monroe 2014). Over the past decades, the programming languages community has made significant progress toward understanding the essence of programming. This paper is an attempt to extend this inquiry to encompass the dynamic evolution of programs. As a first step in this direction, we shall concern ourselves with the question of describing, comparing and computing changes at a syntactical level.

Maintaining a software as complex as an operating system with as many as several thousands contributors is a technical feat made possible thanks, in part, to a venerable Unix utility: `diff` (Hunt and McIlroy 1976). The `diff` tool computes the line-by-line difference between two textual files, determining the smallest set of insertions and deletions of lines to transform one file into the other.

This limited grammar of changes works particularly well for programming languages that organize a program into lines of code. For example, consider the following modification that extends an existing `for`-loop to not only compute the sum of the elements of an array, but also compute their product:

```haskell
sum := 0;
prod := 1;
for (i in is) {
    sum += i;
    prod *= i;
}
```

However, the bias towards lines of code may lead to (unnecessary) conflicts when considering other programming languages. For instance, consider the following diff between two Haskell functions that adds a new argument to an existing function:

```haskell
- head [] = error "?!"
- head (x :: xs) = x
+ head [] = error "Expecting a non-empty list."
+ head (x :: xs) = x
```

This modest change impacts all the lines of the function’s definition.

The line-based bias of the `diff` algorithm may lead to unnecessary conflicts when considering changes made by multiple developers. Consider the following innocuous improvement of the original `head` function, that improves the error message raised when the list is empty:

```haskell
head [] = error "Expecting a non-empty list."
```

Trying to apply the patch above to this modified version of the `head` function will fail, as the lines do not match – even if both changes modify distinct parts of the declaration in the case of non-empty lists.

The `diff` tool is too rudimentary for our purposes. The previous examples suggest that we should exploit the structure of the input files, beyond simple lines of code. This structure can be described by the abstract syntax tree of the language under consideration, or any approximation thereof. These trees can be modelled accurately using algebraic datatypes in a functional language.
2 Background

Before we can introduce our generic notion of patches and the algorithms to compute them, we first introduce some of the terminology regarding patches that we will use throughout this paper and the generic programming technology on which our development relies.

Patches

A patch describes a transformation between two values. We expect any implementation of patches to support at least the following two functions: a diff function that, given two values, creates a patch describing how to transform one into the other; and an apply function that, given a patch and a value, applies the transformation described by this patch to the value if possible. Formally, this specification translates into the following abstract interface:

\[
\text{Patch} : \text{Set} \rightarrow \text{Set} \\
\text{diff} : A \rightarrow A \rightarrow \text{Patch} A \\
\text{apply} : \text{Patch} A \rightarrow A \rightarrow \text{Maybe} A \\
\text{correctness} : \text{apply} (\text{diff} x y) x \equiv (\text{just } y)
\]

Note that applying a patch is a partial operation. A patch that removes a specific line from a file can only be applied to files containing that line; if it is applied to any other file, the result is undefined. The final correctness property states that a patches created through the call \( \text{diff} x y \) should indeed faithfully reconstruct \( y \) from \( x \). By convention, we will refer to the two arguments of the diff function as the source and destination respectively.

Given this specification, there is a trivial implementation of patches that we can define for any type equipped with an equality test:

\[
\text{Patch}_A : \text{Set} \rightarrow \text{Set} \\
\text{diff}_A : A \rightarrow A \rightarrow \text{Patch}_A A \\
\text{diff}_A x y = (x, y) \\
\text{apply}_A : \text{Patch}_A A \rightarrow A \rightarrow \text{Maybe} A \\
\text{apply}_A (x, y) z = \begin{cases} y & \text{if } y \equiv z \text{ then just } y \\ \text{nothing} & \text{else} \end{cases}
\]

Here we model a patch as a pair of the source and destination values. The \( \text{diff}_A \) function is trivial; the \( \text{apply}_A \) function checks whether the argument value \( z \) is equal to the original source value \( x \) and returns the destination \( y \). Note that if source and destination are already equal, then \( \text{apply}_A \) amounts to an identity function, ie. we have \( \text{apply}_A (\text{diff}_A x x) z = z \).

This simple model of patches and application is the one used by most version control systems to manage binary files. However, it is often too simplistic. Even though creating a patch is very efficient, the resulting patches do not contain any meaningful information about the changes that have been made. Furthermore, we can only ever apply a patch to a single value: this is especially problematic when having to reconcile two separate patches to the same value.

For these reasons, any realistic diff algorithm will try to exploit information about the structure of the data being compared. Before we can do so, however, we will set the stage for our algorithm by introducing the generic programming technology upon which it relies.
Datatype Reflection

To compute more accurate patches, we will need to account for more structure than mere lines of text. As a first step in this direction, we shall consider data following a regular tree structure. We model such algebraic datatypes in Agda using universe constructions (Backhouse et al. 1998; Jansson and Jeuring 1997; Martin-Löf 1984; Morris 2007). In particular, we will adopt de Vries and Löh (2014)’s internalization of data-structures as sums-of-products (SoP).

It is straightforward to extend this approach to handle mutual recursion, as hinted at in Section 7.

A universe of sums-of-products. As suggested by the name “sums-of-products”, we will consider finitary coproducts of finitary products of atomic types. An atom can either be some type constant, such as integers or booleans, or a type variable, used to represent recursion.

data Atom : Set where
  K : Konst → Atom
  I : Atom
where Konst is a (finite) enumeration of the constant types available to the programmer.

From our standpoint, constant types are structurally opaque: we can only test them for equality and, therefore, they only support the naive implementation of patches. Throughout this article, we take

data Konst : Set where
  [\ldots]k : Konst → Set
  kN : Konst
  kB : Konst
whereas our formal development is in fact parametrized by any such enumeration.

We then represent the SoP structure using a list of coproducts, each storing a list of products:

data Prod : Set
Prod = List Atom
Sum = List Prod

The type Sum defines the code of our SoP universe. Its interpretation builds its corresponding pattern functor (Benke et al. 2003) of type Set → Set:

\[
\begin{align*}
[\ldots]_a & : \text{Atom} \to (\text{Set} \to \text{Set}) \\
\{I\}_a X &= X \\
\{K \times I\}_a X &= \{K\}_a \times \{I\}_a X \\
[\ldots]_p & : \text{Prod} \to (\text{Set} \to \text{Set}) \\
\{I\}_p X &= X \\
\{\alpha : \pi\}_p X &= \{\alpha\}_p \times \{\pi\}_p X \\
[\ldots]_s & : \text{Sum} \to (\text{Set} \to \text{Set}) \\
\{I\}_s X &= \bot \\
\{\pi : \sigma\}_s X &= \{\pi\}_s \times \{\sigma\}_s X
\end{align*}
\]

Using this universe, we encode the representation of lists of natural numbers and 2-3-trees as

\[
\begin{align*}
\text{listF} : \text{Sum} \\
\text{listF} &= \text{let leafT} = [] \\
& \quad \text{leafT} = [K \text{kN}, 1, 1] \\
& \quad \text{3nodeT} = [K \text{kN}, 1, 1, 1] \\
& \quad \text{in} \ [\text{leafT}, \text{2nodeT}, \text{3nodeT}]
\end{align*}
\]

and we verify that these codes interpret to the expected pattern functors:

\[
\begin{align*}
\text{listF}_{[\ldots]} X &= \text{Unit} \uplus \text{N} \times X \times \text{Unit} \uplus \bot \\
\text{listF}_{[\ldots]} X &= \text{Unit} \uplus \text{N} \times X \times X \times \text{Unit} \uplus \bot
\end{align*}
\]

Despite the redundant unit and \(\bot\) types, these functors are isomorphic to the functors used to represent lists of numbers and 2-3 trees (Gibbons 2007; Yakushev et al. 2009). To make our code more readable, we will omit the functor argument \(X\) and the subscripts \(a, p, s\) in the interpretation function \(\text{typeOf}_{\text{[\ldots]}}\), whenever these may be inferred from the context.

Inhabitants of the SoP universe thus have a canonical representation: they consist of a choice of constructor among the sum of possibilities, applied to a product of its arguments. For a SoP \(\sigma\), choosing a constructor amounts to picking an element in a finite set of length \(\sigma\) elements:

Constr : Sum → Set
Constr \(\sigma\) = Fin (length \(\sigma\))

On our earlier examples, we name the usual constructor tags as follows:

\begin{align*}
\text{con} \text{nil} & : \text{Constr listF} \\
\text{leaf 2-node 3-node} & : \text{Constr 2-3-treeF} \\
\text{nil} &= \text{zero} \\
\text{leaf} &= \text{zero} \\
\text{con} \ &= \text{suc zero} \\
\text{2-node} &= \text{suc zero} \\
\text{3-node} &= \text{suc (suc zero)}
\end{align*}

A view on sums-of-products. Given a constructor, we can extract the type of its arguments using the typeOf function below. Together with suitable arguments, we construct an inhabitant of the desired type with the inj \(_\sigma\) function.

\[
\begin{align*}
\text{typeOf} : (\sigma : \text{Sum}) \to \text{Constr} \sigma \to \text{Prod} \\
\text{typeOf} []) & = [] \\
\text{typeOf} (\pi : \sigma) \text{zero} & = \pi \\
\text{typeOf} (\pi : \sigma) \text{(suc n)} & = \text{typeOf} \sigma n \\
\text{inj}_{\text{con} \text{nil}} & : (C : \text{Constr} \sigma) \to [\text{typeOf} \sigma C]_A \to [\sigma]_A A \\
\text{inj}_{\text{con} \text{zero}} & = \text{inj}_{\text{con} \text{nil}} \\
\text{inj}_{\text{con} \text{suc 0}} & = \text{inj}_{\text{con} \text{nil}} \text{con} \text{suc 0}
\end{align*}
\]

To witness this isomorphism between the encoded representation and the constructor-based presentation, we define a view (McBride and McKinna 2004; Wadler 1987) that telescopes apart any value in our SoP universe as a constructor applied to a list of arguments:
data SOP : ||σ|| X → Set where
  tag : (C : Constr σ) → [||TypeOf σ|| C]p X
  → SOP (injC p)
  sop : ||σ|| Sum (s : ||σ|| X) → SOP s
  sop [] []
  sop [x :: σ] (i1 p) = tag zero p
  sop [x :: σ] (i2 s)
  with sop s
  ... tag C' s' = tag (suc C) C'

  We can use this view, for instance, to implement the dual of
  injC, which will try to pattern-match on constructor i and return
  its arguments, if possible.

match. · : (C : Constr σ) → [||σ|| X → Maybe [||TypeOf σ|| C]p
matchC s with sop s
... | tag C' p with C ≡ C'
... | true = just p
... | false = nothing

Tying the knot. The SoP provides us with a first-order language for
describing pattern functors. To represent actual data-structures, we
must tie the knot and construct the least fixpoint of such functors:

data Fix (σ : Sum) : Set where
(\_ \_ : [||σ|| X (Fix σ) → Fix σ

We can now define the list and 2-3-tree data-structures as follows:

list = Fix listF
2-3-tree = Set
2-3-tree = Fix 2-3-treeF

Moreover, we can also define constructors for list in terms of the
tags we defined earlier:

nil : list
nil = (injF nil tt)
cons : N → list → list
cons x xs = (injF cons (x , xs , tt))

Defining leaf, 2-node and 3-node in a similar way lets one
easily write some values as if they were values of datatypes defined
directly in Agda:
l0 : list
l0 = cons 8 (cons 13 (cons 21 (cons 34 nil)))
l0 : 2-3-tree
l0 = 3-node 1 leaf (2-node 2 leaf leaf) (2-node 3 leaf leaf)

t1 = 2-node 1 (2-node 3 v1 x) y
t2 = 2-node 2 (3-node 3 w1 x' w2) y

These two trees are depicted in Figure 1a and Figure 1c respectively.
The figures show that we will draw our 2-3-trees using black nodes
to represent a 2-node and white nodes to represent a 3-node. We
have replaced the indetermined subtrees (such as x, y and v1) with
labelled leaves in our figures.

How should we represent the transformation mapping t1 into t2?
Traversing the trees from their roots, we see that on the outermost
level they both consist of a 2-node, yet the fields of the source and
destination nodes are different: the first field is modified from a 1 to a 2;
the third field y is the same, and can be copied from the source to
destination, but the second field is modified. Inspecting the second
argument of 2-node in greater detail we can see that at this level, the
constructor has changed from a 2-node to a 3-node. We have many
different ways to continue recursively. In this example, we assume
that the fields of two subtrees are modified in the following way:
the integer 3 is simply copied from the source to the destination;
the source subtree v1 is deleted; the destination subtree w1 and w2 are
inserted; and finally, the source subtree x is modified to construct a
new subtree x'. All of this is depicted graphically in Figure 1b.

This graphical notation enables us to make a couple of observations
before delving into the formal treatment. We will begin by
sketching its intended semantics. First, we remark that the structure
of a patch closely follows the structure of the trees on which it
operates. For instance, the head constructors of t1 and t2 being
identical, the resulting patch should preserve this sharing – as the
figure suggests. Second, when two subtrees are absolutely identical –
as is the case for the subtree y – the resulting patch does not need
to store any data and merely record the fact that this data should
be copied from the source to the destination without modification,
depicted by the cloud that ‘hides’ information about the specific
values involved. Third, being opaque, distinct atoms – such as
the first and third fields of the source and destination values.
This is depicted by the diamond storing both 1 and 2 as the leftmost
child of our patch.

Finally, and more interestingly, a patch may transform one con-
structor into another – as we have seen after matching the outer-
most nodes of t1 and t2. At that point, we need to decide how to
continue transforming the various subtrees of the two values
involved. We will refer to such a choice of association between the
constructor fields of the source value and the constructor fields of
the destination value as an alignment. Here we chose to copy the
value 3, and recursively continue with the two subtrees x and x'.
The other subtrees are either deleted from the source (v1) or
inserted into the destination (w1 and w2). In the figure, we have tried
to visualize this by taking the fields of the 2-node on the left and the
fields of the 3-node on the right. When we choose to associate
two fields from the source and destination, we draw a line between
them. The first and third fields of the source are associated with
the first and third fields of the destination respectively. For each
such association, we either copy the values if they are the same
(such as for 3) or pair them if they are different (such as x and x').
Any remaining subtrees from the source are deleted (as is the case
for v1); any remaining subtrees from the destination are inserted
(as is the case w1 and w2).
This example shows that any definition of patches must handle the coproduct structure and the product structure of our universe separately. The coproduct structure of our patches records changes to constructors; the product structure of our patches records how to align the constructor fields of the coproducts. This distinction is reflected in the definition of patch application, which will follow this phase separation.

Spines

The first part of our algorithm handles the sums of the SoP universe. Given two values, $x$ and $y$, it computes the spine, capturing the common coproduct structure. We distinguish three possible cases:

- $x$ and $y$ are fully equal, in which case we copy the full values regardless of their contents. Figure 2a depicts how an entire subtree $t$ is copied from the source to the destination.
- $x$ and $y$ have the same constructor – i.e., $x = \text{inj}_C px$ and $y = \text{inj}_C py$ – but some subtrees of $x$ and $y$ are distinct, in which case we copy the head constructor and handle all arguments pairwise. Figure 2b illustrates this treatment of values built from equal constructors.
- $x$ and $y$ have distinct constructors, in which case we record a change in constructor and a choice of the alignment of the source and destination’s constructor fields (Figure 2c).

The datatype $S$, defined below, formalizes this description. The three cases we describe above correspond to the three constructors of $S$. When the two values are not equal, we need to represent the differences somehow. If the values have the same constructor (Figure 2b) we need to reconcile the fields of that constructor whereas if the values have different constructors (Figure 2c) we need to reconcile the products that make the fields of the constructors. We parametrized the data type $S$ by a relation between products, $Al$, that describes what to do with the different fields, and a predicate between atoms $At$, that describes what to do with the paired fields in case we have the same constructor.

\begin{align*}
data \ S & \ (At : \text{Atom} \to \text{Set}) \ (Al : \text{Prod} \to \text{Prod} \to \text{Set}) \\
\sigma & : \text{Sum} \\
\text{Scp} & : \ S \ At \ Al \ \sigma \\
\text{Scns} & : \ (C : \text{Constr} \ \sigma) \\
\rightarrow & \ Al \ At \ \text{(typeOf} \ \sigma \ C) \\
\rightarrow & \ S \ At \ Al \ \sigma \\
\text{Schg} & : \ (C_1 \ C_2 : \text{Constr} \ \text{ty}) \\
\rightarrow & \ Al \ (\text{typeOf} \ \text{ty} \ C_1) \ (\text{typeOf} \ \text{ty} \ C_2) \\
\rightarrow & \ S \ At \ Al \ \sigma
\end{align*}

Figure 2. Spines, graphically
The Scp constructor simply records that the tree is unchanged. The Scns constructor records that the first constructor is the same in both trees, so we take the arguments of both subtrees and zip them together. Here the function All lifts the predicate At to work on the fields of both constructors. We omit its definition. Finally, the Schg constructor records a change to the outermost constructor.

Recall the spine shown in Figure 1b. We could represent it using a value of type S as:

\[
S = \text{Scns } 2\text{-node } (p, \text{Schg } 2\text{-node } 3\text{-node } a, \text{Scp})
\]

For p and a with appropriate types.

Note that the AI parameter, which handles products, needs to be heterogeneous: if a constructor changes, the products of arguments may be of different arity and type. The example in Figure 1b shows this situation. There we see that a black node has three fields, whereas a white node has four. As a result, we cannot assume that when a constructor changes, we have exactly the same number of constructor fields. The spine itself, though, is defined for a single type: we are only interested in differing elements of the same type, hence, the outermost coproduct structure shall always be the same.

We gave a very intensional description of what a spine consists of, but we are in fact interested in how to interpret these spines—that is, what do they mean. Below we interpret a spine as a partial function. This is also called the application function of a spine, as it describes the action a spine has on a particular value.

To do so, we require argument functions specifying how to handle both the atoms, doAt, and the underlying product structure, doAl. We proceed by matching the spine and argument tree, yielding the following definition in applicative style (McBride and Paterson 2008).

\[
\text{doAt } \alpha \rightarrow \left[\alpha\right]_a \rightarrow \text{Maybe } \left[\alpha\right]_a
\]
\[
\rightarrow \text{doAl } : \text{AI } \pi_2 \pi_1 \rightarrow \left[\pi_2\right]_p \rightarrow \text{Maybe } \left[\pi_1\right]_p
\]
\[
\rightarrow \text{S At AI } \sigma \rightarrow \left[\sigma\right]_s \rightarrow \text{Maybe } \left[\sigma\right]_s
\]

apply-S doAt doAl Scps x = just x
apply-S doAt doAl Scps i ps x with sop x
...| tag cx dx with cx \equiv i
...| false = nothing
...| true = inj \ circ <S> doAl p dx
apply-S doAt doAl (Scps i ps) x with sop x
...| tag cx dx with cx \equiv i
...| false = nothing
...| true = inj \ circ <S> sAll doAt ps

where

\[
sAll : \text{doAt } \alpha \rightarrow \left[\alpha\right]_a \rightarrow \text{Maybe } \left[\alpha\right]_a
\]
\[
\rightarrow \text{AI } \pi \rightarrow \left[\pi\right]_p \rightarrow \text{Maybe } \left[\pi\right]_p
\]

We distinguish three possible cases. In the first case, Scp, we can copy over the argument tree without inspecting it. In the second case, Schg, we check whether or not the argument is built from the expected constructor. If so, we process the underlying product structure and reassemble a new tree using inj\_i; if not, the application fails and returns nothing. Finally in the case for Scns, we once again check if the argument is built from the constructor we are expecting. If so, we map doAt over the pairs of atoms, corresponding to the constructors field. If this succeeds, we can construct a new tree using inj\_i; if this fails, the entire application returns nothing.

Alignment

Whereas the previous section showed how to match the constructors of two trees, we still need to determine how to continue diffing the products of data stored therein. At this stage in our construction, we are given two heterogeneous lists, corresponding to the fields associated with two distinct constructors. As a result, these lists need not have the same length nor store values of the same type. In the example in Figure 1, we compare two subtrees built from different constructors. To do so, we need to decide how to line up the constructor fields of the source and destination. We shall refer to the process of reconciling the lists of constructor fields as solving an alignment problem.

Our approach is inspired by the existing algorithms computing the edit distance between two strings. When comparing two text files, the diff utility computes an edit script, that is, a sequence of operations that copy lines from the source to the destination file, insert new lines in the destination file, or discard lines from the source file. There are many different choices of edit script for any two files. One extreme example would be to delete all the lines from the source file and insert each line from the destination file. In practice, however, the diff utility minimizes the number of insertions and deletions, copying information whenever it can.

Finding a suitable alignment between two lists of constructor fields amounts to finding a suitable edit script, that relates source fields to destination fields. The AI data type below describes such edit scripts for a heterogeneously typed list of atoms. These scripts may insert fields in the destination (AIns, Figure 3c), delete fields from the source (Adel, Figure 3b), or associate two fields from both lists (AX, Figure 3a). Each of the illustrations in Figure 3 shows how a large alignment (delineated by the dashed lines) can be decomposed into a smaller recursive alignment (delineated by the inner box labelled AI). Depending on whether the alignment associates the heads, deletes from the source list or inserts into the destination, the smaller recursive alignment has shorter lists of constructor fields at its disposal.

data AI (At : Atom \rightarrow Set) : Prod \rightarrow Prod \rightarrow Set where

A0 : \rightarrow \text{Al } []
AX : \alpha \rightarrow \text{AI } \pi_2 \pi_1 \rightarrow \text{Al } (\alpha :: \pi_2) (\alpha :: \pi_1)
Adel : \left[\alpha\right]_a \rightarrow \text{Al } \pi_2 \pi_1 \rightarrow \text{Al } (\alpha :: \pi_2) \pi_1
AIns : \left[\alpha\right]_a \rightarrow \text{Al } \pi_2 \pi_1 \rightarrow \text{Al } \pi_2 (\alpha :: \pi_1)

As we did for spines, we once again abstract over the predicate between the underlying atoms, At : Atom \rightarrow Set.

Note that we require alignments to preserve the order of the arguments of each constructors, thus forbidding permutations of arguments. In effect, the datatype of alignments can be viewed as an intensional representation of (partial) order and type preserving mappings (McBride 2005), mapping source fields to destination fields. Provided a partial embedding for atoms, we can therefore interpret alignments into a function transporting the source fields over to the corresponding destination fields, failure potentially occurring when trying to associate incompatible atoms:

apply-Al (doAt : At \alpha \rightarrow \left[\alpha\right]_a \rightarrow \text{Maybe } \left[\alpha\right]_a)
\rightarrow \text{Al } \pi_2 \pi_1 \rightarrow \left[\pi_2\right]_p \rightarrow \text{Maybe } \left[\pi_1\right]_p


![Alignment Diagrams](image)

(a) $\text{AX}$: sharing of a sub-tree

(b) $\text{Adel}$: deletion of a source sub-tree

(c) $\text{Ains}$: insertion of a destination sub-tree

Figure 3. Alignments, graphically

4 Fixpoint of Changes

In the previous section, we presented patches describing changes to the coproducts, products, and atoms of our $\text{SOP}$ universe. This development, however, was parametrized over the treatment of recursive subtrees: it handles a single layer of the fixpoint construction, but does not yet recurse. In this section, we tie the knot and define patches describing changes to arbitrary recursive datatypes. Throughout the remainder of this section, we will consider patches on a fixed code from our $\text{SOP}$ universe, $\mu\sigma$.

To represent generic patches on recursive datatypes, we will define two mutually recursive data types $\text{Al}$ and $\text{Ctx}$. The semantics of both these datatypes will be given by defining how to apply them to arbitrary values:

- Much like alignments for products, a similar phenomenon appears at fixpoints. When comparing two recursive structures, we can insert, remove or modify constructors. We will use $\text{Al}_\mu : \text{Set}$ to specify these edit scripts at the constructor-level. Its application function has type:
  
  \[
  \text{apply-Al}_\mu : \text{Al}_\mu \rightarrow \text{Fix } \mu\sigma \rightarrow \text{Maybe } (\text{Fix } \mu\sigma)
  \]

- Whenever we choose to insert or delete a recursive subtree, we must specify where this modification takes place. To do so, we will define a new type $\text{Ctx} : \text{Prod } \rightarrow \text{Set}$, inspired by zippers (Huet 1997), to navigate through our data-structures. A value of type $\text{Ctx } \pi$ selects a single atom $l$ from the product of type $\pi$. We can use $\text{Ctx } \pi$ to specify both insertions and deletions:
  
  \[
  \begin{align*}
  \text{insCtx} & : \text{Ctx } \pi \rightarrow \text{Fix } \mu\sigma \rightarrow \text{Maybe } [\pi]_p \\
  \text{delCtx} & : \text{Ctx } \pi \rightarrow [\pi]_p \rightarrow \text{Maybe } (\text{Fix } \mu\sigma)
  \end{align*}
  \]

We will now define both these data types and their associated application functions more precisely.

Aligning Fixpoints

Modeling changes over fixpoints closely follows our definition of alignments of products. Instead of inserting and deleting elements of the product we insert, delete or modify constructors. Our previous definition of spines merely matched the constructors of the source and destination values – but never introduced or removed them. It is precisely these operations that we must account for here.

The $\text{Al}_\mu$ datatype has three constructors: $\text{spn}$, $\text{ins}$, and $\text{del}$:

\[
\begin{align*}
\text{data } \text{Al}_\mu & : \text{Set} \\
\text{spn} & : \text{S } \text{At}_\mu (\text{Al }\text{At}_\mu) \mu\sigma \rightarrow \text{Al}_\mu \\
\text{ins} & : (\text{C } : \text{Constr } \mu\sigma) \rightarrow \text{Ctx } (\text{typeOf } \mu\sigma \text{ C}) \rightarrow \text{Al}_\mu \\
\text{del} & : (\text{C } : \text{Constr } \mu\sigma) \rightarrow \text{Ctx } (\text{typeOf } \mu\sigma \text{ C}) \rightarrow \text{Al}_\mu
\end{align*}
\]
Choosing a Subtree

Our definition of insertion and deletions relies on identifying one recursive argument among the product of possibilities. To model this accurately, we define an indexed zipper to identify a recursive atom (indicated by a value of type \(1\)) amongst a product of atoms:

\[
data \text{Ctx} : \text{Prod} \rightarrow \text{Set} \text{ where}
\]

\[
\begin{align*}
\text{here} & : \text{Al}_\mu \rightarrow \text{All} \varnothing \pi \rightarrow \text{Ctx} \langle 1, \pi \rangle \\
\text{there} & : \varnothing \pi \rightarrow \text{Ctx} \pi \rightarrow \text{Ctx} \langle \alpha, \pi \rangle
\end{align*}
\]

The constructor \(\text{here}\) designates the first subtree; the constructor \(\text{there}\) skips the head and recurses. Note that \(\text{Ctx}\) is defined mutually recursively with \(\text{Al}_\mu\). Whenever we select the subtree on which to recurse, we require a value of type \(\text{Al}_\mu\) describing to proceed.

To complete the definition of patch application given above, we still need to define insertions and deletions on these contexts. To insert a new constructor, we exploit the value of type \(\text{Ctx}\) we are given to determine where to plug in source tree:

\[
\begin{align*}
\text{insCtx} & : \text{Ctx} \pi \rightarrow \text{Fix} \mu \sigma \rightarrow \text{Maybe} \varnothing \pi,
\end{align*}
\]

\[
\text{insCtx} \left( \text{here spmu atmos} \right) x = \langle \_ :: \text{atmos} \rangle \langle \_ :: \_ \rangle \text{apply-At} \text{spmu} x
\]

\[
\text{insCtx} \left( \text{there atmos} \right) x = \langle \_ :: \_ \rangle \langle \_ :: \_ \rangle \text{insCtx} \langle \_ :: \_ \rangle x
\]

Once we have encountered the \(\text{here}\) constructor, we recursively call \(\text{apply-At}\).

Conversely, upon deleting a constructor from the source structure, we exploit \(\text{Ctx}\) to indicate find the subtree that should be used for the remainder of the patch application, discarding all other constructor fields:

\[
\begin{align*}
\text{delCtx} & : \text{Ctx} \pi \rightarrow \varnothing \pi \rightarrow \text{Maybe} \left( \text{Fix} \mu \sigma \right),
\end{align*}
\]

\[
\text{delCtx} \left( \text{here spmu atmos} \right) \left( x :: p \right) = \text{apply-At} \text{spmu} x
\]

\[
\text{delCtx} \left( \text{there atmos} \right) \left( \_ :: p \right) = \text{delCtx} \left( \_ :: p \right)
\]

This deletion function discards any information we have about all the constructor fields, except for the subtree used to continue the patch application process. This greatly increases the domain of the application function. Nonetheless, we store information about these fields using a \(\text{Ctx}\) structure to guarantee that our patches are invertible (Section 6). If we were to discard this information, inverting a deletion to produce an insertion would require us to invent the data with which to populate the remaining constructor fields out of thin air.

Recursive Atoms

Finally, we still need to specify how to handle the atoms. Where our previous definition merely accounted for opaque types, we would like to finally tie the recursive knot. That is, we want to use the datatype \(\text{Al}_\mu\) defined above, whenever we reach a recursive position. The \(\text{At}\) type defined at the end of the previous section abstracted over the treatment of recursive variables. We can instantiate it to use our \(\text{Al}_\mu\) type as follows:

\[
\text{A}_\mu : \text{Atom} \rightarrow \text{Set}
\]

\[
\text{A}_\mu = \text{At} \text{Al}_\mu
\]

Similarly, the application function on atoms at the end of the previous section abstracted over the handling of recursive variables. By passing in the application function defined above, writing \(\text{apply-At} \text{apply-At} \text{Al}_\mu\), we can construct the desired application function on atoms. Finally, putting these definitions together, we obtain the type of patches over our \(\text{SoP}\) universe \(\mu \sigma\):

\[
\text{Patch}_\mu : \text{Set}
\]

\[
\text{Patch}_\mu = \text{Al}_\mu
\]
apply : Patchṣ → Fix μσ → Maybe (Fix μσ)
apply = apply-Alμ

**Patches do not go wrong:** an easily overlooked property of our patch definition is that the destination values it computes are guaranteed to be type-correct by construction. This is unlike the line-based or untyped approaches (which may generate ill-formed values) and similar to earlier results on type-safe differences (Lempsink et al. 2009).

**Meta-programming & programming:** While this paper focuses on the definition and study of the Agda model, the definition of the Patchṣ datatype is also of interest to programmers of the non-dependent kind: we may specialize—through partial evaluation—the definition of Patchṣ to a specific type described by the SoP universe. In this fashion, we obtain a non-dependent, algebraic datatype describing changes for that specific structure. For example, in the following section, we show that the edit scripts generated by Unix diff can be translated to our generic definition.

**Examples**

So far, we have presented a data type to model changes in a structured fashion. Before exploring the search space and showing how one can enumerate patches between two values, we shall illustrate our definitions by considering two small case studies.

**Patches of S-expressions**

To show the full applicability of our approach, let us imagine a simple macro language based on s-expressions, represented by the following datatype:

```
data SExp = N String
           | Lit String
           | Def String SExp SExp
           | SExp :> SExp
           | Nil
```

We have chosen this language to be as simple as possible. A more accurate account of a more realistic programming language would require several (mutually recursive) data types. While our patches and application functions can be extended to handle such datatypes, we refrain from doing so for now.

Let us start by defining the head function, that returns the head of an s-expression or an error code if the expression is nil. Its definition is shown on the left; the corresponding AST, as an inhabitant of SExp, is shown on the right:

```lisp
(defun head (s)  k1 = Def "head" (N "s" => Nil)
  (if (null s)
    (N "if" => (N "null" => N "s" => Nil)
           => (N "error" => Lit "!" => Nil)
           => (N "car" => N "s" => Nil)
    => Nil)
  => error "!"
  => (car s))
)
```

Suppose a programmer decides that the error message is uninformative. We can modify the head function accordingly, call its AST k2:

```lisp
(defun head (s)
  (if (null s)
    (error "empty list")
    (car s))
)
```

We can represent the changes that programmer just made using a Patchṣ, instantiated to work on our SExp datatype. Doing so enables us to define a patch that only modifies the error message, and nothing else. Reusing our suggestive notation from the introduction, we could write:

```lisp
(defun head (s)  (defun head (s)  (fun head (s)
  (if (null s)
    (error (- "!" -)(+ "empty list" +))
    (car s))
)
```

If we compute the patch between these two SExp, this yields a value of type Patchṣ. Applying this patch, is extensionally equal to the following function:

```
(defun head (s)  (defun head (s)  (if (null s)
    (failWith "!"
    => (car s))
  ))
```

Once again we can compute the patch associated with this change. The corresponding application function, app₃, is extensionally equal to the following definition:

```
(defun head (s)  (defun head (s)  (if (null s)
    (failWith "!"
    => (car s))
  ))
```

```
app₁₂ = nothing
```

Here we see that the patch requires the source values to have a certain form. In particular, it maps the string constant "!?" to "empty list".

Interestingly, this patch may also be applied to other values than the original head function defined above. To illustrate this point, consider another modification that might be made to the original head function defined above. Instead of crashing, we might want to raise an exception by calling the function failWith:

```
(defun head (s)  (defun head (s)  (if (null s)
    (failWith "!
    => (car s))
  ))
```

```
app₁₂ = nothing
```

In a line based setting, these two changes would produce a conflict when merged: the same line was edited in two different ways. When considering these modifications, however, only affect distinct SExpS. As a result, our patches can be merged trivially. It is easy to check that app₁₂ ∗ app₁₃ ≡ app₁₃ ∗ app₁₂, that is, applying both patches produces the same head function, regardless of the order in which they are applied:

```
(defun head (s)  (defun head (s)  (if (null s)
    (failWith "empty list"
    => (car s))
  ))
```

**5 Enumerating Patches**

In the previous section, we have devised a typed representation for differences. We have seen that this representation is interesting in and by itself: being richly-structured and typed, it can be thought of...
as a non-trivial programming language whose denotation is given by the app function. However, as programmers, we are mainly interested in computing patches from a source and a destination.

In the following section, we provide a nondeterministic specification of such an algorithm. This approach allows us to remain oblivious to various grammar-specific heuristics we may want to consider in practice, thus focusing our attention on the overall structure of the search space. In particular, we shall strive to identify don’t care nondeterminism – for which all choices lead to a successful outcome – from don’t know nondeterminism – for which a choice may turn out to be incorrect or sub-optimal.

Since we describe our algorithm in Agda, we model don’t know nondeterminism by programming in the List monad. Nondeterministic choice is modelled by list concatenation, which we denote by _ <> _ whereas the absence of valid choice is modelled by the empty list, which we denote by θ.

Computing the Spine Given any two trees, we have seen in Section 3 that a spine represents their longest shared prefix. Computing a spine is thus entirely determined by the source and destination trees, pairing together any distinct subtrees it encounters. We denote the resulting diagonal interpretation _ <> _ by Trivial _ <> _. This notational device allows us to distinguish patches consisting of a pair of a source and a destination value from routine usage of pairs.

spine : [ ] → [ ] → S Trivial2 Trivial2 σ
spine x y with x = y | sop x | sop y...
...true = _ <> _ = Scp
...false | tag cx dx | tag cy dy = if cx ≅ cy
then Sens cx (zip dx dy)
else Schg cx cy (dx , dy)

The call spine x y first checks if x and y are equal. If so, this amounts to performing a blind copy between source and destination.

The constructor change, from the former to the latter, and their respective arguments must be aligned.

Enumerating Alignments Conversely, there are many ways to align two heterogeneous lists in a type-preserving manner. In fact, this is a typed counterpart to the common subsequence problem solved by the Unix diff tool, which tries to minimize the number of insertions or deletions of lines of text.

However, minimizing insertions or deletions in absolute terms may yield sub-optimal results when considering data-structures. Indeed, deleting a large, shared subtree so as to enable the copy of several trivial atoms is unlikely to be more useful than the patch that copies the shared subtree at the expense of some minor modifications. There is a significant body of work studying various edit distances for structured data: we refer our reader to Bille (2005) for a survey of the field. There is however a general trend in adopting structure-specific metrics, depending in the semantics of the data (Autexier 2015). Therefore, rather than commit to a particular edit-distance in the specification of algorithm, we shall consider any valid alignment in a don’t know nondeterministic manner.

The align function thus consists in enumerating all the possible type-preserving edit-scripts for two heterogeneous lists:

align⁺ : { π₂ π₁ : Prod } [π₂]ₚ → [π₁]ₚ → List (Al Trivialₐ π₂ π₁)
align⁺ tt tt = return A0
align⁺ tt (a₂ , p₂) = Ains a₂ <> align⁺ tt p₂
align⁺ (a₁ , p₁) tt = Adel a₁ <> align⁺ p₁ tt
align⁺ (a₁ :: π₂) (a₂ :: π₁) (a₁ , p₁) (a₂ , p₂)
with a₁ ≡ a₂

By focusing on type-preserving edit-scripts, the enumeration function need only consider alignments that relate atoms of the same type. It is nonsensical to even try to relate, say, a boolean with a natural number. Unlike the untyped approach, the type-directed approach enables us to exploit the discriminating power of types to efficiently guide the exploration of the search space.

Provided an enumeration function for type variables, doP, we can enumerate the possible ways of handling atoms:

diff-At : { a : Atom } (doP : X → X → List P) → [σ]ₐ X
          → [σ]ₐ X → List (At P)
diff-At doP [K κ] k₁ k₂ = return (set (k₁ , k₂))
diff-At doP [1] x₁ x₂ = fix <> doP x₁ x₂

We can finally combine all the previous ingredients: enumeration of the spine; enumeration of all their subsequent alignments, and the enumeration over opaque types in order to obtain the enumeration function for the type-preserving functorial alignments:

diff-S : (doP : X → X → List P) → [σ]ₐ X
          → [σ]ₐ X → M (S (At P) (Al (At P)) σ)

Enumerating Recursive Alignments To compute the difference between two recursive structures, we must first establish an alignment of their constructors. There are 3 possible cases: either both source and destination constructors are aligned, in which case we produce a spn code and enumerate the functorial changes (case diff-mod), or a constructor may have to be inserted using an ins code before the source to align with the destination (case diff-ins), or a constructor may have to be deleted from the source using a del code to align with the destination (case diff-del).

diff-Alu : Fix μσ → Fix μσ → List Alμ
diff-Alu (x) (y) = diff-mod x y
<=| diff-ins (x) y
<=| diff-del x (y)

Enumerating the functorial alignments is already handled by the function diff-S, introduced in the previous section. Deleting or inserting a constructor is fully determined by the source or the destination data-structure:

diff-del : [μσ]ₐ → Fix μσ → List Alμ
diff-del s₁ x₂ with sop s₁
... | tag C₁ p₁ = del C₁ <> diff-Ctx x₂ p₁
Nonetheless, this refinement is but a first step toward a practical realization. Being implemented in a pure type theory, it does not exploit programming techniques such as memoization to tame don’t know nondeterminism nor concurrency to exploit don’t care nondeterminism. By exploiting standard techniques for reflecting efficient optimization and proof search techniques in type theory (Claret et al. 2013; Tristan and Leroy 2008), we can nonetheless have the best of both worlds: our universe of changes may be understood as a certificate language, witnessing the existence of a valid patch. The production of such a certificate may be left to any (non-verified, imperative) program.

6 Structure and Interpretation of patches

One motivation for adopting structured differences is to improve the accuracy and compositionality of patches, thus increasing their versatility. Indeed, while we compute patches between two versions of a structured document, we intend to apply them to potentially modified versions of that document and to combine them into coherent patches. The granularity of Unix diff being at the line level, the resulting patches are line-accurate: if a patch identifies a difference on a given line, any subsequent modification on that line – however orthogonal – will trigger a conflict that requires manual intervention. Intuitively, structured patches enable the description of more accurate modifications: the unit of change is at level of an atom of the grammar while conflicts may only occur on insertion and deletion points, that is, precisely those points where we actually inspect our input during patch application. In the following, we set this intuition on a formal footing by characterising accuracy of application functions.

Accurate Interpretation. We have seen that, extensionally, patches correspond to partial functions (Section 4). We will refer to one patch as being more accurate than another if it succeeds in producing a patched result more often. Formally, this amounts to lifting the canonical partial order on Maybe \( A \)

\[
\begin{align*}
\text{nothing} \leq y \\
\text{just } x & \leq \text{just } y \text{ iff } x \equiv y
\end{align*}
\]

to functions by pointwise comparison over the input domain, i.e., we have \( f \leq g \) if \( \forall x . f x \leq g x \). This corresponds to the usual extension order (Robinson and Rosolini 1988) of partial maps.

While this definition faithfully translates our intuition, it is unsatisfactory from a programming standpoint: it is too extensional to be actionable. We could certainly use it for proofs, but would be out of luck to use it for programming. Instead, we would like to exploit the syntactic nature of structured patches to compute a measure of accuracy akin to the edit distance in the string alignment problem. We could then relate – once and for all – that measure with its extensional specification introduced above. The idea is to compute a natural number, called the cost, for every patch. The validity of such a cost function is then established with respect to the extension order: given two candidates patches \( p \) and \( q \) from \( x \) to \( y \), it should be the case that if \( p \leq q \), then \( \text{apply } q \leq \text{apply } p \).

Groupoid Structure. Throughout Section 3 and Section 4, we have been careful to give a perfectly symmetric representation of differences. As a result, we noticed that, in the del case for example, the patch records information that is not strictly necessary for implementing the apply function. Maintaining this symmetry is however crucial to enable us to implement patch inversion, \( \text{inv} : \)
Patch \rightarrow \text{Patch}, which transforms a patch from source $x$ to destination $y$ into a patch from $y$ to $x$. The implementation of this operation is unsurprising: it merely transforms insertions into deletions, and visa versa, while going through the symmetrical constructs of all our patch operations.

In Section 4, we have seen that, in some cases, we can successfully apply two patches consecutively, in any order. To do so, we worked at the semantic level and composed the (partial) apply functions of both patches. This led us to investigate whether a similar notion of patch composition exists at the syntactic level. This amounts to defining a partial composition operator, $\text{cmp} : \text{Patch} \rightarrow \text{Patch} \rightarrow \text{Maybe Patch}$, tentatively producing a patch combining the effects of both of its inputs. Partiality comes from the fact that two patches may make incompatible changes to the input, such as modifying an opaque value in two inconsistent ways. Applying such patches would also never produce a valid result.

Interestingly, the combination of a total identity function and a partial composition function yields a groupoid structure, which brings our constructions under the comfortable umbrella of previous developments in patch theory (Angiuli et al. 2014; Jacobson 2009). Our work can thus be seen as an intensional presentation of these general results for regular data-structures. By offering the desired groupoid structure, it combines with those more general frameworks, which may provide support for handling binary files or file-system management in an orthogonal manner.

### 7 Related work

The diffing problem can be portrayed in a variety of different flavors. The untyped approach has been thoroughly studied in both its linear (Bergroth et al. 2000) and tree (Akutsu et al. 2010; Autexier 2015; Bille 2005; Chawathe and Garcia-Molina 1997; Demaine et al. 2007; Klein 1998) variations. The canonical solution for the untyped linear scenario is the well-known Unix `diff` (Hunt and McIlroy 1976). For the tree-structured variation, though, a variety of implementations (Falleri et al. 2014; Farinier et al. 2015; Hashimoto and Mori 2008) has arisen in the last few years. In this paper, however, we have explored how to exploit the type `structure` of trees to give a more precise account of our diff algorithm.

Several other pieces of related work exploring the possibilities of defining a data type generic diff algorithm, most notably the work by Lempsink et al. (Lempsink et al. 2009) and Vassena (Vassena 2014). Both define a data type generic program to compute a diff of structured data. Their algorithm, however, fundamentally differs from the one presented here. Both papers extend the linear diff algorithm, as used by the Unix `diff` utility, to structured data by considering the pre-order traversal of a data type. This flattening of the tree structure makes reasoning about patches especially hard.

Beyond diffing, there is a great deal of work on version control systems. The canonical example of a formal VCS is Darcs (Roundy 2005). 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 (Jacobson 2009). Another example is the Pijul VCS, inspired by Mimram (Mimram and Giusto 2013). This uses category theory to define and reason about patches. The base category on which their work is built, however, handles files as a list of lines, thus providing only a theoretical framework on top of the already existing Unix `diff`. Swierstra and Löh (Swierstra and Löh 2014) apply separation logic and Hoare calculus to be able to define a logic for reasoning about patches. Separation logic is particularly useful to prove the disjoinedness of patches – and guarantee that their associated apply functions commute. Finally, Angiuli et al. (Angiuli et al. 2014) have developed a model of patch theory within Homotopy Type Theory. Although their model considers various patches and repositories, it does not provide a generic account for arbitrary data types as done here.

### Discussion

For the sake of simplicity and uniformity, we have focused our attention on simple regular types in this paper. Our Agda model, however, supports mutually recursive definitions as well, enabling us to represent and deal with more practical examples of programming languages. As a result, the recursive alignment (Section 4) becomes heterogeneous since we may need to align constructors of distinct inductive types. For example, in a rose tree, we may want to insert a rose tree into a list of rose trees, leading to an heterogeneous alignment between rose tree and list of rose tree.

Our presentation was also made easier by a casual treatment of recursion, especially in Section 3 where we focused on the functional semantics. As a result, when tying the knot in Section 4, our Agda model cannot automatically check for termination for, say, the application function. We are convinced that we could write a single mutually-recursive definition, at the expense of significantly obfuscating the code. We refrained from doing so, noticing that termination follows trivially from the fact that our definitions merely structurally recur over the `Patchy` datatype.

### Future Work

There are several directions for future work that we have already started exploring. Although we have used Agda to explore our definition of patches, the `sums-of-products` universe we have chosen is still fairly simple. We would like to extend it to also incorporate the dependently typed `sigma-of-sigmas` universe, used to model dependent inductive types (Chapman et al. 2010).

Our experiments so far have confirmed our belief that the algorithms and the framework presented here give a more accurate account of patches for tree structured data than simple line based diffs. To provide further evidence to support this, however, we intend to specialize our algorithm to work on the abstract syntax trees of a specific language. We could then replay the merge history of large source code repositories online; this would give further empirical evidence to support our claim.

In this paper we have not yet considered the question of merging patches: given two patches to the same source value, when can they be merged into a single patch? This process is of paramount importance to consolidate changes from different collaborators, as is done by modern version control systems.

### Conclusion

Defining a generic diff algorithm between algebraic data types is no easy task. Each component of the datatype – the choice of constructor, the constructor fields, and recursive subtrees – may all change in different ways. The purpose of this paper is to account for all these changes in an accurate and generic fashion. Defining the type of patches and their semantics, however, is a crucial first step in the larger research program that we envision. With this definition in hand, we can further explore the theoretical underpinnings, such as the algebra of generic patches. There are also immediate practical applications: instantiating our algorithm to a specific datatype allows us to collect empirical results about avoiding unnecessary conflicts in practice. None of this work is possible, however, without the results presented here.
References


