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-evariste.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

ACM Reference format:
Victor Cacciari Miraldo, Pierre-Évariste Dagand, and Wouter Swierstra. 2017. Type-Directed Diffing of Structured Data. In *Proceedings of 2nd ACM SIGPLAN International Workshop on Type-Driven Development, Oxford, UK, September 3, 2017 (TyDe'17),* 14 pages.
DOI: 10.1145/3122975.3122976

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 an 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:

```
    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:

```
- head []       = error "?!"
- head (x :: xs) = x
+ head []       d = d
+ head (x :: xs) d = 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:

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

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

TyDe'17, Oxford, UK

978-1-4503-5183-6/17/09…$15.00
DOI: 10.1145/3122975.3122976

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.

If we reconsider our example, we could give a more accurate description of the modification made to the *head* function by describing the changes made to the constituent declarations and expressions:

```
head []        {+d+} = error {-"?!"-} {+"Expect..."+}
head (x :: xs) {+d+} = x
```

There is more structure here than mere lines of text. In this paper we will show how to exploit this.

We will present a conceptual and practical framework for defining, specifying and manipulating transformations between structured data. We take advantage of dependent type theory as a uniform programming language for (meta-)programming typed representations of data as well as a formal system for specifying algorithms. We will extensively program *with* dependent types to ensure, by construction, that our transformations are structure-preserving. In particular, this paper makes the following contributions:

- We define a generic notion of patch for structured data. Our definition is *structure and type preserving*: applying patches will always produce well-formed data. We will initially present a solution for computing the difference between non-recursive datatypes (Section 3), before extending our results to arbitrary algebraic datatypes (Section 4). We illustrate our approach by revisiting the previous examples and showing that it subsumes the notion of edit script used by the Unix `diff` utility.
- We provide a nondeterministic specification of the differencing algorithm (Section 5) computing structured patches. This specification provides a blueprint for the implementation of domain-specific heuristics by precisely identifying the backtracking points guiding the search space exploration. Furthermore, our specification is executable: we can enumerate all possible patches between two inputs. We illustrate the practical benefit of this formal framework in two different ways. Firstly, we show that – due to sharing and type information – non-trivial patches can be successfully applied to values besides their original input. This is especially important when considering our patches as the potential basis for a version control system, where modifications may be interleaved in a non-trivial fashion. Secondly, we implement a domain-specific heuristic for a data model exhibiting unique labels, which improves the accuracy of the resulting patches.
- Finally, we explore the structure and interpretation of our type-directed patches (Section 6). We shall reveal the groupoid structure of patches by defining inversion and composition as patch manipulating programs. The interpretation of patches as partial functions enables us to formalize a notion of "patch accuracy".

Throughout this paper we will use Agda (Norell 2009) to model increasingly complex patches. We will sometimes simplify the code presented in this paper, for example, by omitting the placement of universal quantifiers and type signatures when these can be inferred from the context. Our entire development is available online.[1]

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:

$$\mathsf{Patch} : \mathsf{Set} \to \mathsf{Set}$$
$$\mathsf{diff} : A \to A \to \mathsf{Patch}\ A$$
$$\mathsf{apply} : \mathsf{Patch}\ A \to A \to \mathsf{Maybe}\ A$$
$$\mathsf{correctness} : \mathsf{apply}\ (\mathsf{diff}\ x\ y)\ x \equiv \mathsf{just}\ y$$
$$\mathsf{identity} : \mathsf{apply}\ (\mathsf{diff}\ x\ x)\ z \equiv \mathsf{just}\ z$$

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 correctness property states that a patches created through the call diff x y should indeed faithfully reconstruct y from x. Moreover, the identity patch should act as the identity on data. 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:

$$\mathsf{Patch}_\Delta : \mathsf{Set} \to \mathsf{Set}$$
$$\mathsf{Patch}_\Delta\ A = A \times A$$
$$\mathsf{diff}_\Delta : A \to A \to \mathsf{Patch}_\Delta\ A$$
$$\mathsf{diff}_\Delta\ x\ y = (x , y)$$
$$\mathsf{apply}_\Delta : \mathsf{Patch}_\Delta\ A \to A \to \mathsf{Maybe}\ A$$
$$\mathsf{apply}_\Delta\ (x , y)\ z = \mathsf{if}\ x \stackrel{?}{=} y\ \mathsf{then}\ \mathsf{just}\ z\ \mathsf{else\ if}\ x \stackrel{?}{=} z\ \mathsf{then}\ \mathsf{just}\ y\ \mathsf{else}\ \mathsf{nothing}$$

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

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

[1] goo.gl/SY2Dj0

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
  kℕ : Konst
  kB : Konst
```

```
⟦_⟧k : Konst → Set
⟦kℕ⟧k = ℕ
⟦kB⟧k = Bool
```

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:

```
Prod : Set
Prod = List Atom
```

```
Sum : Set
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:

```
⟦_⟧a : Atom → (Set → Set)
⟦I⟧a   X = X
⟦K κ⟧a X = ⟦κ⟧k
```

```
⟦_⟧p : Prod → (Set → Set)
⟦[]⟧p     X = Unit
⟦α :: π⟧p X = ⟦α⟧a X × ⟦π⟧p X
```

```
⟦_⟧s : Sum → (Set → Set)
⟦[]⟧s X       = ⊥
⟦π :: σ⟧s X = ⟦π⟧p X ⊎ ⟦σ⟧s X
```

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

```
listF : Sum
listF = let nilT  = []
            consT = [ K kℕ , I ]
        in [nilT , consT]
```

```
2-3-treeF : Sum
2-3-treeF = let leafT  = []
                2nodeT = [ K kℕ , I , I ]
                3nodeT = [ K kℕ , I , I , I ]
            in [leafT , 2nodeT , 3nodeT ]
```

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

```
⟦listF⟧s : Set → Set
⟦listF⟧s X = Unit ⊎ ℕ × X × Unit ⊎ ⊥
```

```
⟦2-3-treeF⟧s : Set → Set
⟦2-3-treeF⟧s X = Unit ⊎ ℕ × X × X × Unit
                 ⊎ ℕ × X × X × X × Unit ⊎ ⊥
```

Despite the redundant Unit and ⊥ 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 $[\![ty]\!]_{\square}\ X$, 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* σ, choosing a constructor amounts to picking an element in a finite set of length σ elements:

```
Constr : Sum → Set
Constr σ = Fin (length σ)
```

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

```
`cons `nil : Constr listF
`nil       = zero
`cons      = suc zero
```

```
`leaf `2-node `3-node : Constr 2-3-treeF
`leaf   = zero
`2-node = suc zero
`3-node = suc (suc zero)
```

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. function.

```
typeOf : (σ : Sum) → Constr σ → Prod
typeOf []       ()
typeOf (π :: σ) zero    = π
typeOf (π :: σ) (suc n) = typeOf σ n
```

```
inj_ : (C : Constr σ) → ⟦typeOf σ C⟧p A → ⟦σ⟧s A
inj_zero   p = i1 p
inj_(suc i) p = i2 (inj_i p)
```

To witness this isomorphism between the encoded representation and the constructor-based presentation, we define a *view* (McBride and McKinna 2004; Wadler 1987) that teases apart any value in our *SoP* universe as a constructor applied to a list of arguments:

```
data SOP : ⟦σ⟧s X → Set where
  tag : (C : Constr σ) (p : ⟦typeOf σ C⟧p X)
      → SOP (inj_C p)
sop : {σ : Sum} (s : ⟦σ⟧s X) → SOP s
sop {[]} ()
sop {π :: σ} (i1 p) = tag zero p
sop {π :: σ} (i2 s)
  with sop s
...| tag C s′ = tag (suc C) s′
```

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

```
match_·_ : (C : Constr σ) → ⟦σ⟧s X → Maybe ⟦typeOf σ C⟧p
match_C 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⟧s (Fix σ) → Fix σ
```

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

```
list : Set
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 = ⟨inj·nil tt⟩
cons : ℕ → list → list
cons x xs = ⟨inj·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)))
t0 : 2-3-tree
t0 = 3-node 1 leaf (2-node 2 leaf leaf) (2-node 3 leaf leaf)
```

3 Functorial Changes

In Section 2, we specified patches through an abstract interface and presented a simple implementation. In the following sections, we will give an intensional model of such a theory for the data-structures described by the *SoP* universe. To do so, we first consider a single layer of datatype, *ie.* a single application of the datatypes pattern functor. In Section 4, we extend this treatment to recursive datatypes, essentially by taking the fixpoint of the constructions we present in this section. Throughout this section, we consider two structured values:

```
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 v_1) with labelled leaves in our figures.

How should we represent the transformation mapping t_1 into t_2? 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 v_1 is *deleted*; the destination subtree w_1 and w_2 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 t_1 and t_2 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 *1* in t_1 and *2* in t_2 – cannot be decomposed further. Instead, we record their difference using the a trivial patch we saw in the previous section, *ie.* as a pair of 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 constructor into another – as we have seen after matching the outermost nodes of t_1 and t_2. 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 (v_1) or inserted into the destination (w_1 and w_2). 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 v_1); any remaining subtrees from the destination are inserted (as is the case w_1 and w_2).

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. Namelly, *Spines* and *Alignments* will be used to handle changes on the sums and products of the universe, respectively.

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 largest 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 – *ie.* $x = \mathsf{inj}_C\ px$ and $y = \mathsf{inj}_C\ py$ – 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 *At*, that describes what to do with the paired fields in case we have the same constructor.

```
data S (At : Atom → Set) (Al : Prod → Prod → Set)
       (σ : Sum) : Set where
  Scp  : S At Al σ
  Scns : (C : Constr σ)
       → All At (typeOf σ C)
       → S At Al σ
  Schg : (C₁ C₂ : Constr ty)
       → Al (typeOf ty C₁) (typeOf ty C₂)
       → S At Al σ
```

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 ommit 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:

$$s = \mathsf{Scns}\ \mathsf{2\text{-}node}\ (p\ ,\ \mathsf{Schg}\ \mathsf{2\text{-}node}\ \mathsf{3\text{-}node}\ a\ ,\ \mathsf{Scp})$$

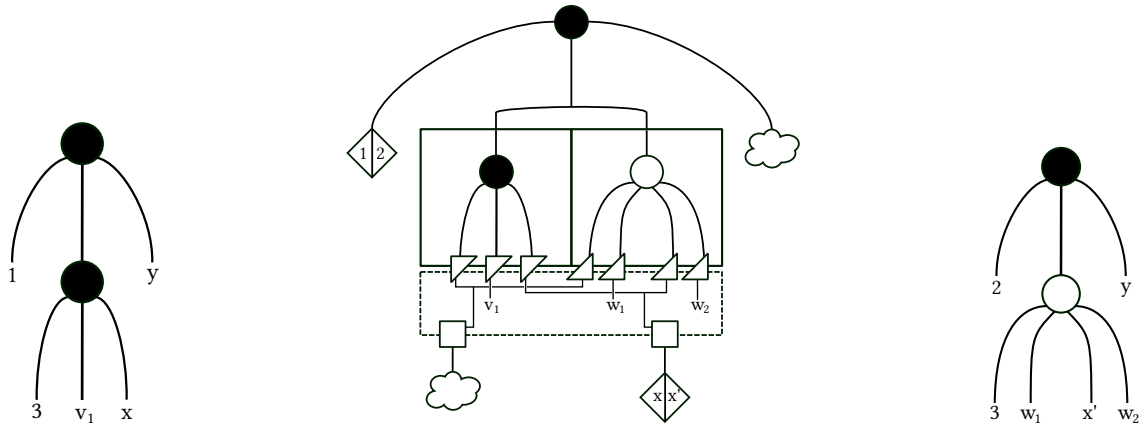
For p and a with appropriate types.

Note that the *Al* parameter, which handles *products*, needs to be a *relation*: 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 diffing 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).

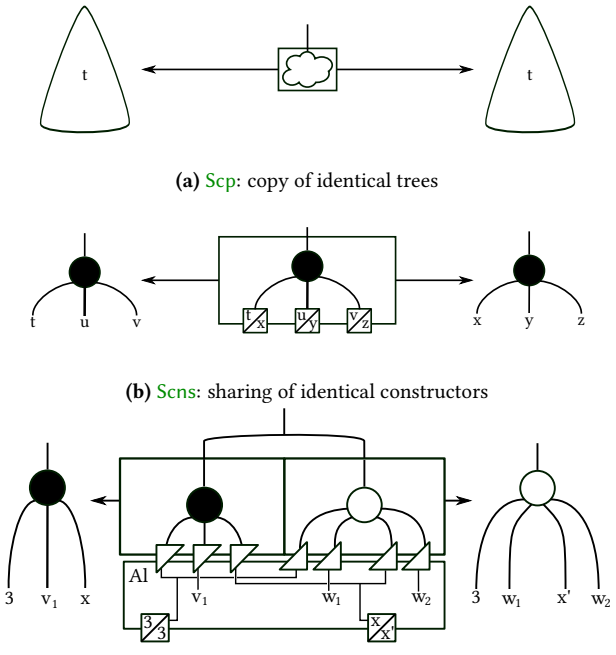
```
apply-S :  (doAt : At α → ⟦α⟧ₐ → Maybe ⟦α⟧ₐ)
        → (doAl : Al π₂ π₁ → ⟦π₂⟧ₚ → Maybe ⟦π₁⟧ₚ)
        → S At Al σ → ⟦σ⟧ₛ → Maybe ⟦σ⟧ₛ
apply-S doAt doAl Scp x = just x
apply-S doAt doAl (Schg i j p) x with sop x
...| tag cx dx with cx ≟ i
...| false      = nothing
...| true       = inj_j <$> doAl p dx
apply-S doAt doAl (Scns i ps) x with sop x
...| tag cx dx with cx ≟ i
...| false      = nothing
...| true       = inj_i <$> sAll doAt ps
  where
    sAll : (doAt : At α → ⟦α⟧ₐ → Maybe ⟦α⟧ₐ)
         → All At π → ⟦π⟧ₚ → Maybe ⟦π⟧ₚ
```



(a) t_1 **(b)** Structured diff between t_1 and t_2 **(c)** t_2

Figure 1. Two trees, *t1* and *t2*, and their difference.

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_j; 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.



(a) Scp: copy of identical trees

(b) Scns: sharing of identical constructors

(c) Schg: change of distinct constructors

Figure 2. Spines, graphically

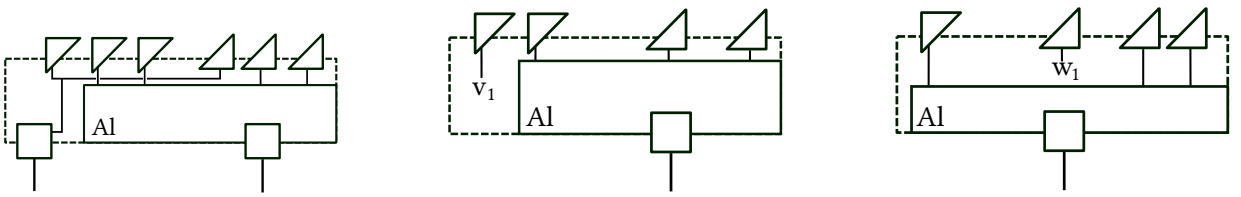
Alignment

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 Al 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 Al). 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 Al (At : Atom → Set) : Prod → Prod → Set where
  A0  :                          → Al At []       []
  AX  : At α → Al At π₂ π₁       → Al At (α :: π₂) (α :: π₁)
```



(a) AX: sharing of a sub-tree **(b)** Adel: deletion of a source sub-tree **(c)** Ains: insertion of a destination sub-tree

Figure 3. Alignments, graphically

```
Adel : ⟦α⟧a → Al At π2 π1 → Al At (α :: π2)      π1
Ains : ⟦α⟧a → Al At π2 π1 → Al At      π2  (α :: π1)
```

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

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 maps*, along the lines of McBride's order preserving embeddings (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 α → ⟦α⟧a → Maybe ⟦α⟧a)
         → Al At π2 π1 → ⟦π2⟧p → Maybe ⟦π1⟧p

apply-Al doAt A0 tt = just tt
apply-Al doAt (AX p al) (a , ps) with doAt p a | apply-Al doAt al ps
...| just a' | just p' = just (a' , p')
...| _       | _       = nothing
apply-Al doAt (Ains a al) ps with apply-Al doAt al ps
...| just ps' = just (a , ps')
...| nothing  = nothing
apply-Al doAt (Adel a1 al) (a2 , ps) with a1 ≟ a2
...| false = nothing
...| true  = apply-Al doAt al ps
```

The definition is unremarkable. It proceeds by straightforward induction on the alignment. When the alignment specifies that a source field is inserted, we perform the remaining alignment, before inserting the new field. When an existing field should be deleted, we need to check that current the value at the 'head' of the list of constructor fields equals the value we are expecting. If so, we delete it and recurse; otherwise the application fails.

Atoms

Having dealt with the coproduct structure through the spine S, and with the product structure therein through the alignments Al, we are left with atoms. For the moment, we focus on non-recursive datatypes and parameterized our construction with respect to a type $P : Set$ of patches for the recursive argument $Rec : Set$. In later sections, we will show how to tie this recursive knot.

```
data At (P : Set) : Atom → Set where
  set : Trivial_K κ → At P (K κ)
  fix : P           → At P I
```

On opaque types, we follow the trivial patch definition of (2.1), define $Trivial_K$ as the diagonal interpretation of the constant types. Recall that $Patch_\Delta\ A = A \times A$.

```
Trivial_K : Konst → Set
Trivial_K κ = Patch_Δ ⟦κ⟧k

apply-K : Trivial_K κ → ⟦κ⟧k → Maybe ⟦κ⟧k
apply-K = apply_Δ
```

Now, given an application function for P, we can define an application function for atoms:

```
apply-At : (doP : P → X → Maybe X) → At P α → ⟦α⟧a
         → Maybe ⟦α⟧a X
apply-At doP (set k) x = apply-K k x
apply-At doP (fix p) x = doP p x
```

In the following section, we will instantiate this parameter P to the handle data types recursively.

4 Fixpoint of Changes

In the previous section, we presented patches describing changes to the coproducts, products, and atoms of our *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 *SoP* universe, $\mu\sigma$.

To represent generic patches on values of Fix $\mu\sigma$, we will define two mutually recursive data types $Al\mu$ and 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 phenomenom appears at fixpoints. When comparing two recursive structures, we can insert, remove or modify constructors. We will use $Al\mu : Set$ to specify these edit scripts at the constructor-level. Its application function has type:
  `apply-Alμ : Alμ → Fix μσ → Maybe (Fix μσ)`

- 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 Ctx : Prod → Set, inspired by zippers (Huet 1997), to navigate through our data-structures. A value of type Ctx π selects a single atom I from the product of type π. We can use Ctx π to specify both insertions and deletions:

```
insCtx : Ctx π → Fix μσ → Maybe ⟦π⟧ₚ
delCtx : Ctx π → ⟦π⟧ₚ   → Maybe (Fix μσ)
```

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 Alμ datatype has three constructors: spn, ins, and del:

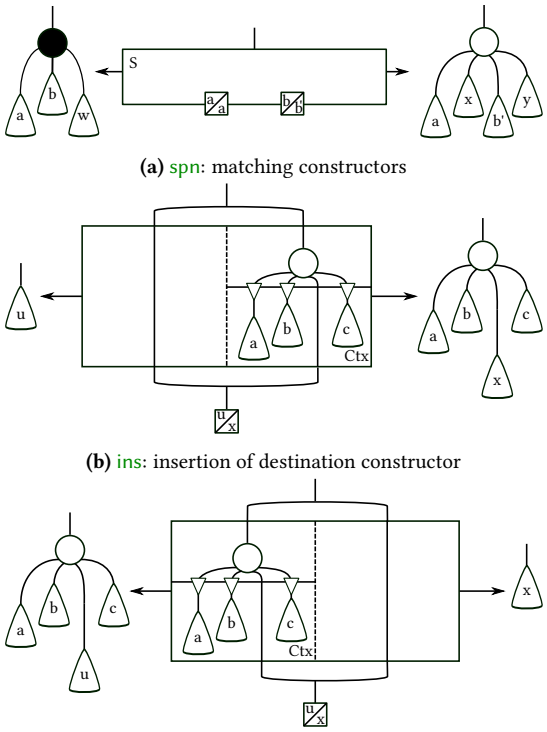
```
data Alμ : Set where
  spn : S Atμ (Al Atμ) μσ → Alμ
  ins : (C : Constr μσ)   → Ctx (typeOf μσ C) → Alμ
  del : (C : Constr μσ)   → Ctx (typeOf μσ C) → Alμ
```

The first constructor, spn, does not perform any new insertions and deletions, but instead records a spine and an alignment of the underlying product structure. This closely follows the patches we have seen in the previous section. To insert a new constructor, ins, requires two pieces of information: a choice of the new constructor to be introduced, C, and the fields associated with that constructor. Note that we only need to record *all but one* of the constructor's fields, as represented by a value of type Ctx (typeOf $\mu\sigma$ C). We have tried to visualize this situation in Figure 4b: to transform the tree u on the left into the larger tree on the right, we introduce a new 3-node. The information stored in the patch records *how* to associate u with one of the subtrees of the new node, together with the remaining constructor fields (a,b, and c). The situation for deletion in Figure 4c is analogous.

The interpretation of these codes is given below:

```
apply-Alμ : Alμ → ⟦σ₁⟧ → Maybe ⟦σ₂⟧
apply-Alμ (spn s) x = apply-S (apply-Al doAt) s x
apply-Alμ (ins C ∂) x with insCtx ∂ x
... | nothing = nothing
... | just x′ = ⟨inj_C x′⟩
apply-Alμ (del C ∂) ⟨x⟩ with sop x
... | tag C₁ p₁ with C ≟ C₁
... | false     = nothing
... | true      = delCtx ∂ p₁
```

In the spn case, we call the application functions defined for spines and alignments that we saw previously. Most of the hard work handling deletions and insertions is done by the insCtx and delCtx, that we shall cover shortly.



(a) spn: matching constructors

(b) ins: insertion of destination constructor

(c) del: deletion of source constructor

Figure 4. Recursive alignments, graphically

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 I) amongst a product of atoms:

```
data Ctx : Prod → Set where
  here  : Alμ   → All ⟦_⟧ₐ π → Ctx (I , π)
  there : ⟦α⟧ₐ → Ctx π       → Ctx (α , π)
```

The constructor here designates the first subtree; the constructor there skips the head and recurses. Note that Ctx is defined mutually recursively with Alμ. Whenever we select the subtree on which to recurse, we require a value of type Alμ describing to 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 Ctx we are given to determine where to plug in source tree:

```
insCtx : Ctx π → Fix μσ → Maybe ⟦π⟧ₚ
insCtx (here spmu atmus) x = (_ :: atmus) <$> apply-Alμ spmu x
insCtx (there atmu ∂)    x = (atmu :: _)  <$> insCtx ∂ x
```

Once we have encountered the here constructor, we recursively call apply-Alμ.

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

```
delCtx : Ctx π → ⟦π⟧ₚ → Maybe (Fix μσ)
delCtx (here spmu atmus) (x :: p)  = apply-Alμ spmu x
delCtx (there atmu ∂)    (at :: p) = delCtx ∂ p
```

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 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 Alμ defined above, whenever we reach a recursive position. The At type defined at the end of the previous section abstracted over the treatment of recursive variables. We can instantiate it to use our Alμ type as follows:

```
At𝜇 : Atom → Set
At𝜇 = At Al𝜇
```

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 apply-At apply-Alμ, we can construct the desired application function on atoms. Finally, putting these definitions together, we obtain the type of patches over our *SoP* universe $\mu\sigma$:

```
Patch𝜇 : Set
Patch𝜇 = Al𝜇

apply : Patch𝜇 → 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 Patchμ datatype is also of interest to programmers of the non-dependent kind: we may specialize – through partial evaluation – the definition of Patchμ 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 a simple case studie.

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. Its primitives are either names (N) or literals. 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:

```
(defun head (s)
  (if (null s)
      (error "!?")
      (car s)
  ))
```

k1 =

```
Def "head" (N "s" ▹ Nil)
           (N "if" ▹ (N "null" ▹ N "s" ▹ Nil)
                   ▹ (N "error" ▹ Lit "!?" ▹ Nil)
                   ▹ (N "car" ▹ N "s" ▹ Nil)
                   ▹ Nil)
```

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

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

We can represent the changes that programmer just made using a Patchμ, 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:

```
(defun head (s)
  (if (null s)
      (error {- "!?" -}{+ "empty list" +})
      (car s)
  ))
```

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

```
app₁₂ : SExp → Maybe SExp
app₁₂ (Def x y (z₁ ▹ z₂ ▹ (w₁ ▹ Lit "!?" ▹ w₂) ▹ z₃))
        = just (Def x y (z₁ ▹ z₂
                        ▹ (w₁ ▹ Lit "empty list" ▹ w₂)
                        ▹ z₃))
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)
  (if (null s)
      (failWith "!?")
      (car s)
  ))
```

Once again we can compute the patch associated with this change. The corresponding application function, app_{13}, is extensionally equal to the following definition:

```
app₁₃ : SExp → Maybe SExp
app₁₃ (Def x y (z₁ ▹ z₂ ▹ (N "error" ▹ w₁ ▹ w₂) ▹ z₃))
        = just (Def x y (z₁ ▹ z₂
                         ▹ (N "failWith" ▹ w₁ ▹ w₂)
                         ▹ z₃))
app₁₃ _ = 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 $\mathsf{app}_{12} \bullet \mathsf{app}_{13} \equiv \mathsf{app}_{13} \bullet \mathsf{app}_{12}$, that is, applying both patches produces the same head function, regardless of the order in which they are applied:

```
(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 $\emptyset$.

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 $[\![\cdot]\!]_\square \times [\![\cdot]\!]_\square$ by $\mathsf{Trivial}_\square \cdot \cdot$. 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 : ⟦σ⟧ → ⟦σ⟧ → S Trivial_a Trivial_p σ
spine x y with x ≟ y | sop x | sop y
...| true  | _          | _          = Scp
...| false | tag cx dx | tag cy dy = if cx ≟ cy
                                     then Scns cx (zip_p 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. If the constructors are equal, then their respective arguments shall be compared pairwise. If the constructors are distinct, we record 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 Trivial_a 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* {α₁ :: π₂} {α₂ :: π₁} (a₁ , p₁) (a₂ , p₂)
  with α₁ ≟ α₂
...| true  = AX (a₁ , p₁) <$> align* p₁      p₂
           <|> Adel a₁     <$> align* p₁ (a₂ , p₂)
           <|> Ains a₂     <$> align* (a₁ , p₁) p₂
...| false = Adel a₁     <$> align* p₁ (a₂ , p₂)
           <|> Ains a₂     <$> align* (a₁ , p₁) p₂
```

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 : {α : Atom} (doP : X → X → List P) → ⟦α⟧ₐ X
        → ⟦α⟧ₐ X → List (At P)
diff-At doP {K κ} k₁ k₂ = return (set (k₁ , k₂))
diff-At doP {I}   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⟧ₛ X → ⟦σ⟧ₛ X → List (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) using diff-S, 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-Alμ : Fix μσ → Fix μσ → List Alμ
diff-Alμ ⟨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⟧ₛ → Fix μσ → List Alμ
diff-del s₁ x₂ with sop s₁
... | tag C₁ p₁ = del C₁ <$> diff-Ctx x₂ p₁

diff-ins : Fix μσ → ⟦μσ⟧ₛ → List Alμ
diff-ins x₁ s₂ with sop s₂
... | tag C₂ p₂ = ins C₂ <$> diff-Ctx x₁ p₂
```

These alignment problems thus reduce to enumerating the well-typed recursive spines, using diff-Ctx, to locate the modification.

Enumerating Subtree Choice We have seen that, conceptually, a recursive spine is a (typed) one-hole context for I atoms: enumerating all valid spines simply amounts to enumerating all the cursors on those I atoms.

```
diff-Ctx : Fix μσ → ⟦π⟧ₚ → List (Ctx π)
diff-Ctx          x₁ [] = ∅
diff-Ctx {K _ :: _} x₁ (k₂ :: ats₂)
  = there k₂ <$> diff-Ctx x₁ ats₂
diff-Ctx {I   :: _} x₁ (x₂ :: ats₂)
  = flip here ats₂ <$> diff-Alμ x₁ x₂
  <|> there x₂ <$> diff-Ctx x₁ ats₂
```

We thus obtain a complete nondeterministic specification of a differencing algorithm, as described by diff-Alμ. In practice, enumerating *all* possible patches is far too expensive due to the exponential nature of the enumeration. Instead, heuristics can be used to favor some patches over others. However, unlike the line-oriented approach, there is no commonly agreed upon differencing heuristic for structured data. In the following section, we consider how grammar-specific heuristics can be developed in our framework.

Heuristics A differencing heuristic can be understood as a more efficient refinement of the nondeterministic specification given above. It takes advantage of domain-specific knowledge to improve accuracy on idiomatic modifications or prune the search space.

Consider the grammar of S-expressions SExp. We may simplify and thus speed up the alignment problem by relying on the name of Def-nodes, demarcating a function's body. If two Def-nodes do not have the same name, we will consider them to be necessarily distinct. This allows us to rule out the possible modification of one into the other. We would then write the following, domain-specific refinement of diff-H:

```
diff-H : SExp → SExp → List (Patchμ SExp SExp)
diff-H ⟨x⟩ ⟨y⟩ with δ (fun-name ⟨x⟩ , fun-name ⟨y⟩)
...| nothing = diff-ins ⟨x⟩ y
             <|> diff-del x ⟨y⟩
...| just (nx , ny)
  with nx ≟ ny
...| false   = diff-ins ⟨x⟩ y
             <|> diff-del x ⟨y⟩
...| true    = diff-mod x y

fun-name : SExp → Maybe String
δ        : Maybe A × Maybe B → Maybe (A × B)
```

Crucially this definition *only* considers the modification alternative, diff-mod, when both functions have the same name. In practice, relying on the function name never changing may not be a good idea – but this small example serves to illustrate how we *might* use such information, rather than prescribing any particular choice.

Nonetheless, this refinement is but a first step toward a practical implementation: being implemented in a pure type theory, it cannot exploit programming techniques such as memoization to tame *don't know* nondeterminsm 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 patchsets. 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

$$\begin{aligned} \mathsf{nothing} &\preceq y \\ \mathsf{just}\ x &\preceq \mathsf{just}\ y \text{ iff } x \equiv y \end{aligned}$$

to functions by pointwise comparison over the input domain, *ie.* we have $f \preceq g$ if $\forall x; f\ x \preceq g\ x$. 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 a 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 $cost\ p \leqslant cost\ q$, then $\mathsf{apply}\ q \preceq \mathsf{apply}\ p$.

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, $inv : \mathsf{Patch} \to \mathsf{Patch}$, 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, $cmp : \mathsf{Patch} \to \mathsf{Patch} \to \mathsf{Maybe\ Patch}$, 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 2016). 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 *disjointedness* of patches – and guarantee that their associated apply functions commute. Finally, Anguiuli 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 a relation 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 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 functorial 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 recurse over the Patchμ 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

Tatsuya Akutsu, Daiji Fukagawa, and Atsuhiro Takasu. 2010. Approximating Tree Edit Distance through String Edit Distance. *Algorithmica* 57, 2 (2010), 325–348. DOI: http://dx.doi.org/10.1007/s00453-008-9213-z

Carlo Angiuli, Edward Morehouse, Daniel R. Licata, and Robert Harper. 2014. Homotopical Patch Theory. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming (ICFP '14)*. ACM, New York, NY, USA, 243–256.

Serge Autexier. 2015. Similarity-Based Diff, Three-Way Diff and Merge. *Int. J. Software and Informatics* 9, 2 (2015), 259–277. http://www.ijsi.org/ch/reader/view_abstract.aspx?file_no=i217

Roland Backhouse, Patrik Jansson, Johan Jeuring, and Lambert Meertens. 1998. Generic programming. In *International School on Advanced Functional Programming*. Springer, 28–115.

Marcin Benke, Peter Dybjer, and Patrik Jansson. 2003. Universes for generic programs and proofs in dependent type theory. *Nordic Journal of Computing* 10 (2003).

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

Philip Bille. 2005. A survey on tree edit distance and related problems. *Theor. Comput. Sci* 337 (2005), 217–239.

James Chapman, Pierre-Évariste Dagand, Conor McBride, and Peter Morris. 2010. The gentle art of levitation. In *International Conference on Functional Programming*. 3–14. DOI: http://dx.doi.org/10.1145/1863543.1863547

Sudarshan S. Chawathe and Hector Garcia-Molina. 1997. Meaningful Change Detection in Structured Data. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data (SIGMOD '97)*. ACM, New York, NY, USA, 26–37. DOI: http://dx.doi.org/10.1145/253260.253266

Guillaume Claret, Lourdes del Carmen González Huesca, Yann Régis-Gianas, and Beta Ziliani. 2013. Lightweight proof by reflection using a posteriori simulation of effectful computation. In *International Conference on Interactive Theorem Proving*. Springer, 67–83.

Michael Cochez, Ville Isomöttönen, Ville Tirronen, and Jonne Itkonen. 2013. The Use of Distributed Version Control Systems in Advanced Programming Courses. In *Proceedings of the 9th International Conference on ICT in Education, Research and Industrial Applications: Integration, Harmonization and Knowledge Transfer, Kherson, Ukraine, June 19-22, 2013*. 221–235. http://ceur-ws.org/Vol-1000/ICTERI-2013-p-221-235.pdf

Edsko de Vries and Andres Löh. 2014. True Sums of Products. In *Proceedings of the 10th ACM SIGPLAN Workshop on Generic Programming (WGP '14)*. ACM, New York, NY, USA, 83–94. DOI: http://dx.doi.org/10.1145/2633628.2633634

Erik D. Demaine, Shay Mozes, Benjamin Rossman, and Oren Weimann. 2007. An Optimal Decomposition Algorithm for Tree Edit Distance. In *Proceedings of the 34th International Colloquium on Automata, Languages and Programming (ICALP 2007)*. Wroclaw, Poland, 146–157.

Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-grained and accurate source code differencing. In *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*. 313–324. DOI: http://dx.doi.org/10.1145/2642937.2642982

Benjamin Farinier, Thomas Gazagnaire, and Anil Madhavapeddy. 2015. Mergeable persistent data structures. In *Vingt-sixièmes Journées Francophones des Langages Applicatifs (JFLA 2015)*, David Baelde and Jade Alglave (Eds.). Le Val d'Ajol, France. https://hal.inria.fr/hal-01099136

Jeremy Gibbons. 2007. Datatype-generic Programming. In *Proceedings of the 2006 International Conference on Datatype-generic Programming (SSDGP'06)*. Springer-Verlag, Berlin, Heidelberg, 1–71. http://dl.acm.org/citation.cfm?id=1782894.1782895

Masatomo Hashimoto and Akira Mori. 2008. Diff/TS: A Tool for Fine-Grained Structural Change Analysis. In *Proceedings of the 2008 15th Working Conference on Reverse Engineering (WCRE '08)*. IEEE Computer Society, Washington, DC, USA, 279–288. DOI: http://dx.doi.org/10.1109/WCRE.2008.44

Gerard Huet. 1997. The zipper. *Journal of functional programming* 7, 5 (1997), 549–554.

J. W. Hunt and M. D. McIlroy. 1976. *An Algorithm for Differential File Comparison*. Technical Report CSTR 41. Bell Laboratories, Murray Hill, NJ.

Judah Jacobson. 2009. A formalization of darcs patch theory using inverse semigroups. *Available from ftp://ftp.math.ucla.edu/pub/camreport/cam09-83.pdf* (2009).

Patrik Jansson and Johan Jeuring. 1997. PolyP—a polytypic programming language extension. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 470–482.

Philip N. Klein. 1998. Computing the Edit-Distance Between Unrooted Ordered Trees. In *Proceedings of the 6th Annual European Symposium on Algorithms (ESA '98)*. Springer-Verlag, London, UK, UK, 91–102.

Greg Kroah-Hartman. 2016. Linux Kernel Development. (2016). https://github.com/gregkh/kernel-development/

Eelco Lempsink, Sean Leather, and Andres Löh. 2009. Type-safe Diff for Families of Datatypes. In *Proceedings of the 2009 ACM SIGPLAN Workshop on Generic Programming (WGP '09)*. ACM, New York, NY, USA, 61–72.

Tommy MacWilliam. 2013. Incorporating Version Control into Programming Courses (Abstract Only). In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education (SIGCSE '13)*. ACM, New York, NY, USA, 759–759. DOI: http://dx.doi.org/10.1145/2445196.2445508

Per Martin-Löf. 1984. *Intuitionistic Type Theory*. Bibliopolis Napoli.

Conor McBride. 2005. *Epigram: Practical Programming with Dependent Types*. Springer Berlin Heidelberg, Berlin, Heidelberg, 130–170. DOI: http://dx.doi.org/10.1007/11546382_3

Conor McBride and James McKinna. 2004. The View from the Left. *J. Funct. Program.* 14, 1 (Jan. 2004), 69–111. DOI: http://dx.doi.org/10.1017/S0956796803004829

Conor McBride and Ross Paterson. 2008. Applicative programming with effects. *Journal of functional programming* 18, 01 (2008), 1–13.

Samuel Mimram and Cinzia Di Giusto. 2013. A Categorical Theory of Patches. *CoRR* abs/1311.3903 (2013).

Don Monroe. 2014. A New Type of Mathematics? *Commun. ACM* 57, 2 (Feb. 2014), 13–15. DOI: http://dx.doi.org/10.1145/2557446

Peter Morris. 2007. *Constructing Universes for Generic Programming*. Ph.D. Dissertation. University of Nottingham.

Ulf Norell. 2009. Dependently typed programming in Agda. In *Advanced Functional Programming*. Springer Berlin Heidelberg, 230–266.

E. Robinson and G. Rosolini. 1988. Categories of partial maps. *Information and Computation* 79, 2 (1988), 95 – 130. DOI: http://dx.doi.org/10.1016/0890-5401(88)90034-X

David Roundy. 2005. Darcs: Distributed Version Management in Haskell. In *Proceedings of the 2005 ACM SIGPLAN Workshop on Haskell (Haskell '05)*. ACM, New York, NY, USA, 1–4. DOI: http://dx.doi.org/10.1145/1088348.1088349

Wouter Swierstra and Andres Löh. 2014. The Semantics of Version Control. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward! '14)*. 43–54.

Jean-Baptiste Tristan and Xavier Leroy. 2008. Formal Verification of Translation Validators: A Case Study on Instruction Scheduling Optimizations. In *Proceedings*

of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '08). ACM, New York, NY, USA, 17–27. DOI: http://dx.doi.org/10.1145/1328438.1328444

The Univalent Foundations Program. 2013. *Homotopy Type Theory: Univalent Foundations of Mathematics*. https://homotopytypetheory.org/book, Institute for Advanced Study.

Marco Vassena. 2016. Generic Diff3 for Algebraic Datatypes. In *Proceedings of the 1st International Workshop on Type-Driven Development (TyDe 2016)*. ACM, New York, NY, USA, 62–71.

P. Wadler. 1987. Views: A Way for Pattern Matching to Cohabit with Data Abstraction. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '87)*. 307–313. DOI: http://dx.doi.org/10.1145/41625.41653

Alexey Rodriguez Yakushev, Stefan Holdermans, Andres Löh, and Johan Jeuring. 2009. Generic programming with fixed points for mutually recursive datatypes. In *Proceeding of the 14th ACM SIGPLAN international conference on Functional programming, ICFP 2009, Edinburgh, Scotland, UK, August 31 - September 2, 2009*. 233–244. DOI: http://dx.doi.org/10.1145/1596550.1596585