Structure aware version control

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

Abstract

stuff

Categories and Subject Descriptors D.1.1 [*look*]: for—this

General Terms Haskell

Keywords Haskell

1. Introduction

Version control has become an indispensable tool in the development of modern software. There are various version control tools freely available, such as git or mercurial, that are used by thousands of developers worldwide. Collaborative repository hosting websites, such as GitHub and Bitbucket, haven triggered a huge growth in open source development. [Add citations]

Yet all these tools are based on a simple, line-based diff algorithm to detect and merge changes made by individual developers. While such line-based diffs generally work well when monitoring source code, not all data is best represented as lines of text.

Consider the following example CSV file, recording the marks and names three students:

```
Student name, Student number, Mark
Alice   , 440, 7.0
Bob     , 593, 6.5
Carroll , 168, 8.5
```

A new entry to this CSV file will not modify any existing entries and is unlikely to cause conflicts. Adding a new column storing the date of the exam, however, will change every line of the file and therefore will conflict with any other change to the file. Conceptually, however, this seems wrong: adding a column changes every line in the file, but leaves all the existing data unmodified. The only reason that this causes conflicts is that the *granularity of change* that version control tools use is not suitable for this kind of file.

This paper proposes a different approach to version control systems. Instead of relying on a single line-based diff algorithm, we will explore how to define a *generic* notion of change, together with algorithms for observing and combining such changes. To this end, this paper makes the following novel contributions:

- We define a *type-indexed* data type for representing edits to structured data. The structure of the data closely resembles the way one defines data-types in functional languages such as Haskell.
- We define generic algorithms for computing and applying a diff generically and prove they are correct with respect to each other.
- A notion of residual is used to propagate changes of different diffs, hence providing a bare mechanism for merging and conflict resolution.
- We illustrate how these ideas in Agda have been implemented in a prototype Haskell tool, capable of automatically merging changes to structured data. This tool provides the user with the ability to define custom conflict resolution strategies when merging changes to structured data. [Cloning and swapping]

Background

- *Should we have this section? It cold be nice to at least mention the edit distance problem and that in the untyped scenario, the best running time is of $O(n^3)$. Types should allow us to bring this time lower.*

2. Structural Diffing

- *Diffing and tree-edit distance are very closely related problems.*
- *This should go on background, though.*

To Research!

- *The LCS problem is closely related to diffing. We want to preserve the LCS of two structures! How does our diffing relate? Does this imply maximum sharing?*
  - ***ANS:** No! We don't strive for maximum sharing. We strive for flexibility and customization. See refactoring*

To make version control to be structure aware we need to define what structures we can handle. In our case, the structure is the universe of context free types, patching will be defined for its inhabitants.

2.1 Context Free Datatypes

The universe of *context-free types*, in the sense of [**?**], is constructed following the grammar CF of context free types with a deBruijn representation for variables.

$$\mathrm{CF} ::= 1 \mid 0 \mid \mathrm{CF} \times \mathrm{CF} \mid \mathrm{CF} + \mathrm{CF} \mid \mu\ \mathrm{CF} \mid \mathbb{N} \mid (\mathrm{CF}\ \mathrm{CF}) \mid \mathrm{CFCF}$$

In Agda, the CF universe is defined by:

```
data U : ℕ → Set where
  u0  : {n : ℕ} → U n
  u1  : {n : ℕ} → U n
  _⊕_ : {n : ℕ} → U n → U n → U n
  _⊗_ : {n : ℕ} → U n → U n → U n
  β   : {n : ℕ} → U (suc n) → U n → U n
  μ   : {n : ℕ} → U (suc n) → U n
  vl  : {n : ℕ} → U (suc n)
  wk  : {n : ℕ} → U n → U (suc n)
```

In order to make life easier we use a deBruijn indexing for our type variables. an element of U n reads as a type with n free type variables. $u0$ and $u1$ represent the empty type and unit, respectively. Products and coproducts are represented by _⊗_ and _⊕_. Recursive types are created through μ. Type application is denoted by β. To control and select variables we use constructors that retrieve the *top* of the variable stack, vl, and that *pop* the variable stack, ignoring the top-most variable, wk. We could have used a Fin to identify variables, and have one instead of two constructors for variables, but that would trigger more complicated definitions later on.

Stating the language of our types is not enough. We need to specify its elements too, after all, they are the domain which we seek to define our algorithms for! Defining elements of fixed-point types make things a bit more complicated, check [**?**] for a more in-depth explanation of these details. We need a decreasing Telescope of types in order to specify the elements of U and still pass the termination checker. The reason for having this telescope is simple. Imagine we want to describe the elements of a type with n variables, ty : U n. We can only speak about this type once all n variables are bound to correspond to a given type. We need then $t_1, t_2, \cdots, t_n$ to pass as *arguments* to ty. Moreover, these types have to have less free variables then ty itself, otherwise this will never finish. This list of types with decreasing type variables is defined through Tel:

```
data Tel : ℕ → Set where
  tnil  : Tel 0
  tcons : {n : ℕ} → U n → Tel n → Tel (suc n)
```

A value (v : El U {n} ty t) reads roughly as: a value of type ty with n variables, applied to telescope t. At this point we can define the actual v's that inhabit every code in U n. In Agda, the elements of U are defined by:

```
data El : {n : ℕ} → U n → Tel n → Set where
  void : {n : ℕ}{t : Tel n}
       → El u1 t
  inl  : {n : ℕ}{t : Tel n}{a b : U n}
       (x : El a t) → El (a ⊕ b) t
  inr  : {n : ℕ}{t : Tel n}{a b : U n}
       (x : El b t) → El (a ⊕ b) t
  _,_  : {n : ℕ}{t : Tel n}{a b : U n}
       → El a t → El b t → El (a ⊗ b) t
  top  : {n : ℕ}{t : Tel n}{a : U n}
       → El a t → El vl (tcons a t)
  pop  : {n : ℕ}{t : Tel n}{a b : U n}
       → El b t → El (wk b) (tcons a t)
  mu   : {n : ℕ}{t : Tel n}{a : U (suc n)}
       → El a (tcons (μ a) t) → El (μ a) t
  red  : {n : ℕ}{t : Tel n}{F : U (suc n)}{x : U n}
       → El F (tcons x t)
       → El (β F x) t
```

The set El of the elements of U is straight forward. We begin with some simple constructor to handle finite types: void, inl, inr and _, _. Next, we define how to reference variables using pop and top. Finally, mu and red specify how to handle recursive types and type applications.

Let us see a simple example of how types and elements are defined in this framework. Consider that we want to encode the list $(u : [\,]) :: [\,U\,]$, for U being the unit type with the single constructor u. We start by defining the type of lists, this is an element of U (suc n), which later lets us define an element of that type.

```
list : {n : ℕ} → U (suc n)
list = μ (u1 ⊕ wk vl ⊗ vl)

myList : {n : ℕ}{t : Tel n} → El list (tcons u1 t)
myList = mu (inr (pop (top void) , top (mu (inl void))))
```

Up until now we showed how to define the universe of context free types and the elements that inhabit it. We are, however, interested in the operations can we perform on these elements. As we shall see, this choice of universe turns out to be very expressive, providing a plethora of interesting operations. The first very usefull concept is the decidability of generic equality[**?**].

```
_≟-U_ : {n : ℕ}{t : Tel n}{u : U n}(x y : El u t) → Dec (x ≡ y)
```

But only comparing things will not get us very far. We need to be able to inspect our elements generically. A very natural way to do so is to see an element of a given type as a tree. We can define operations like getting the list of immediate children, or computing their arity, that is, how many children do they have. Such operations, coupled with generic decidable equality, allows us to traverse and compare two arbitrary trees.

```
children : {n : ℕ}{t : Tel n}{a : U (suc n)}{b : U n}
  → El a (tcons b t) → List (El b t)

arity : {n : ℕ}{t : Tel n}{a : U (suc n)}{b : U n}
  → El a (tcons b t) → ℕ
```

The advantage of doing so in Agda, is that we can prove that our definitions are correct.

```
children-arity-lemma
  : {n : ℕ}{t : Tel n}{a : U (suc n)}{b : U n}
  → (x : El a (tcons b t))
  → length (children x) ≡ arity x
```

We can even go a step further and say that every element is defined by a constructor and a vector of children, with the correct arity. This lets us treat generic elements as elements of a (typed) rose-tree, whenever thas is convenient.

```
unplug : {n : ℕ}{t : Tel n}{a : U (suc n)}{b : U n}
  → (el : El a (tcons b t))
  → Σ (El a (tcons u1 t)) (λ x → Vec (El b t) (arity x))

plug : {n : ℕ}{t : Tel n}{a : U (suc n)}{b : U n}
  → (el : El a (tcons u1 t))
  → Vec (El b t) (arity el)
  → El a (tcons b t)

plug-correct : {n : ℕ}{t : Tel n}{a : U (suc n)}{b : U n}
  → (el : El a (tcons b t))
  → el ≡ plug (p1 (unplug el)) (p2 (unplug el))
```

These operations are central to solving the diffing problem, for we need a way of traversing generic trees in order to decide how to change one into the other. We do things a bit differently, though. We linearize the trees and proceed to traverse the resulting list.

For the readers familiar with Haskell's *Uniplate*[**?**] library, our plug and unplug operations allow for a similar view over datatypes. For instance, we can define the transform function in Agda as:

```
transform : {n : ℕ}{t : Tel n}{ty : U n}
  → (f : {n : ℕ}{t : Tel n}{ty : U n} → ElU ty t → ElU ty t)
  → ElU ty t → ElU ty t
transform {t = tnil} f el
  = f el
transform {t = tcons x t} f el
  = let hdE , chE = unplug el
    in f (plug hdE (V.map (transform f) chE))
```

Note that we first need to pattern-match on the telescope, as types with zero variables can not be *unpluged*.

- *Vassena's and Loh's universe is the typed rose-tree! Correlate!!*

This repertoire of operations, and the hability to inspect an element structurally, according to its type, gives us the toolset we need in order to start describing differences between elements. That is, we can now start discussing what does it mean to *diff* two elements or *patch* an element according to some description of changes.

2.2 Patches over a Context Free Type

A patch over T is an object that describe possible changes that can be made to objects of type T. The high-level idea is that diffing two objects $t_1, t_2 : T$ will produce a patch over T, whereas applying a patch over T to an object will produce a $Maybe\ T$. It is interesting to note that application can not be made total. Let's consider $T = X + Y$, and now consider a patch $(Left\ x) \xrightarrow{p} (Left\ x')$. What should be the result of applying p to a $(Right\ y)$? It is undefined!

The type of *diff*'s is defined by D. It is indexed by a type and a telescope, which is the same as saying that we only define *diff*'s for closed types[1]. However, it also has a parameter A, this will be addressed later.

```
data D {a}(A : {n : ℕ} → Tel n → U n → Set a)
  : {n : ℕ} → Tel n → U n → Set a where
```

As we mentioned earlier, we are interested in analizing the set of possible changes that can be made to objects of a type T. These changes depend on the structure of T, for the definition follows by induction on it.

If T is the Unit type, we can not modify it.

```
D-void : {n : ℕ}{t : Tel n} → D A t u1
```

If T is a product, we need to provide *diffs* for both its components.

```
D-pair : {n : ℕ}{t : Tel n}{a b : U n}
  → D A t a → D A t b → D A t (a ⊗ b)
```

If T is a coproduct, things become slighlty more interesting. There are four possible ways of modifying a coproduct, which are defined by:

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

[1] Types that do not have any free type-variables

Let us take a closer look at the diffs of a coproduct. There are two options we can take when modifying a coproduct $a \oplus b$. Given some diff p over a, we can always modify things that inhabit the left of the coproduct by D-inl p. Or we can change some given value left x into a right y, this is captured by D-setl x y. The case for D-inr and D-setr are analogous.

We also need some housekeeping definitions to make sure we handle all types defined by U.

```
D-β : {n : ℕ}{t : Tel n}{F : U (suc n)}{x : U n}
  → D A (tcons x t) F → D A t (β F x)
D-top : {n : ℕ}{t : Tel n}{a : U n}
  → D A t a → D A (tcons a t) vl
D-pop : {n : ℕ}{t : Tel n}{a b : U n}
  → D A t b → D A (tcons a t) (wk b)
```

Fixed points are handled by a list of *edit operations*. We will discuss them in detail later on.

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

The aforementioned parameter A goes is used in a single consrtuctor, allowing us to have a free-monad structure over D's. This shows to be very usefull for adding extra information, as we shall discuss, on section 3.1, for adding conflicts.

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

Finally, we define Patch t ty as D $(\lambda \_\ \_ \to \bot)$ t ty. Meaning that a Patch is a D with *no* extra information.

2.3 Producing Patches

Given a generic definition of possible changes, the primary goal is to produce an instance of this possible changes, for two specific elements of a type T. We shall call this process *diffing*. It is important to note that our gdiff function expects two elements of the same type! This constrasts with the work done by Vassena[**?**] and Lempsink[**?**], where their diff takes objects of two different types.

For types which are not fixed points, the gdiff functions follows the structure of the type:

```
gdiff : {n : ℕ}{t : Tel n}{ty : U n}
  → ElU ty t → ElU ty t → Patch t ty
gdiff {ty = vl} (top a) (top b)      = D-top (gdiff a b)
gdiff {ty = wk u} (pop a) (pop b)    = D-pop (gdiff a b)
gdiff {ty = β F x} (red a) (red b) = D-β (gdiff a b)
gdiff {ty = u1} void void = D-void
gdiff {ty = ty ⊗ tv} (ay , av) (by , bv)
  = D-pair (gdiff ay by) (gdiff av bv)
gdiff {ty = ty ⊕ tv} (inl ay) (inl by) = D-inl (gdiff ay by)
gdiff {ty = ty ⊕ tv} (inr av) (inr bv) = D-inr (gdiff av bv)
gdiff {ty = ty ⊕ tv} (inl ay) (inr bv) = D-setl ay bv
gdiff {ty = ty ⊕ tv} (inr av) (inl by) = D-setr av by
gdiff {ty = μ ty} a b = D-mu (gdiffL (a :: []) (b :: []))
```

Where the gdiffL takes care of handling fixed point values. The important remark here is that it operates over lists of elements, instead of single elements. This is due to the fact that the children of a fixed point element is a (possibly empty) list of fixed point elements.

Recursion Fixed-point types have a fundamental difference over the other type constructors in our universe. They can grow or shrink arbitrarily. We have to account for that when tracking differences between their elements. As we mentioned earlier, the diff of a fixed point is defined by a list of *edit operations*.

```
data Dμ {a}(A : {n : ℕ} → Tel n → U n → Set a)
  : {n : ℕ} → Tel n → U (suc n) → Set a where
```

But the interesting bits are the *edit operations* we allow, where Val a t = El U a (tcons u1 t):

```
Dμ-ins : {n : ℕ}{t : Tel n}{a : U (suc n)}
  → ValU a t → Dμ A t a
Dμ-del : {n : ℕ}{t : Tel n}{a : U (suc n)}
  → ValU a t → Dμ A t a
Dμ-dwn : {n : ℕ}{t : Tel n}{a : U (suc n)}
  → D A (tcons u1 t) a → Dμ A t a
```

Again, we have a constructor for adding *extra* information, which is ignored in the case of Patches.

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

The edit operations we allow are very simple. We can add or remove parts of a fixed-point or we can modify non-recursive parts of it. and instead of copying, we introduce a new constructor, Dμ-dwn, which is responsible for traversing down the type-structure. Copying is modelled by Dμ-dwn (gdiff x x). The intuition is that for every object x there is a diff that does not change x, we will look into this on section 2.3.

Before we delve into diffing fixed point values, we need some specialization of our generic operations for fixed points. Given that $\mu X.F\ X \approx F\ 1 \times$ List $(\mu X.F\ X)$, that is, any inhabitant of a fixed-point type can be seen as a non-recursive head and a list of recursive children. We then make a specialized version of the plug and unplug functions, which lets us open and close fixed point values.

```
Openμ : {n : ℕ} → Tel n → U (suc n) → Set
Openμ t ty = El U ty (tcons u1 t) × List (El U (μ ty) t)


μ-open : {n : ℕ}{t : Tel n}{ty : U (suc n)}
  → El U (μ ty) t → Openμ t ty


μ-close : {n : ℕ}{t : Tel n}{ty : U (suc n)}
  → Openμ t ty → Maybe (El U (μ ty) t × List (El U (μ ty) t))
```

Although the plug and unplug uses vectors, to remain total functions, we drop that restriction and switch to lists instead, this way we can easily construct a fixed-point with the beginning of the list of children, and return the unused children, this will be very conveient when defining how patches are applied. Nevertheless, a soundness lemma guarantees the correct behaviour.

```
μ-close-resp-arity
  : {n : ℕ}{t : Tel n}{ty : U (suc n)}{a : El U (μ ty) t}
    {hdA : El U ty (tcons u1 t)}{chA l : List (El U (μ ty) t)}
  → μ-open a ≡ (hdA , chA)
  → μ-close (hdA , chA ++ l) ≡ just (a , l)
```

We denote the first component of an *opened* fixed point by its *value*, or *head*; whereas the second component by its children. The diffing of fixed points, which was heavily inspired by [**?**], is then defined by:

```
gdiffL : {n : ℕ}{t : Tel n}{ty : U (suc n)}
  → List (El U (μ ty) t) → List (El U (μ ty) t) → Patchμ t ty
gdiffL [] [] = []
gdiffL [] (y :: ys) with μ-open y
...| hdY , chY = Dμ-ins hdY :: (gdiffL [] (chY ++ ys))
gdiffL (x :: xs) [] with μ-open x
...| hdX , chX = Dμ-del hdX :: (gdiffL (chX ++ xs) [])
gdiffL (x :: xs) (y :: ys)
  = let
      hdX , chX = μ-open x
      hdY , chY = μ-open y
      d1 = Dμ-ins hdY :: (gdiffL (x :: xs) (chY ++ ys))
      d2 = Dμ-del hdX :: (gdiffL (chX ++ xs) (y :: ys))
      d3 = Dμ-dwn (gdiff hdX hdY) :: (gdiffL (chX ++ xs) (chY ++ ys))
  in d1 ⊔μ d2 ⊔μ d3
```

The first three branches are simple. To transform [ ] into [ ], we do not need to perform any action; to transform [ ] into $y : ys$, we need to insert the respective values; and to transform $x{:}xs$ into [ ] we need to delete the respective values. The interesting case happens when we want to transform $x : xs$ into $y : ys$. Here we have three possible diffs that perform the required transformation. We want to choose the diff with the least *cost*, for we introduce an operator to do exactly that:

```
_⊔μ_ : {n : ℕ}{t : Tel n}{ty : U (suc n)}
  → Patchμ t ty → Patchμ t ty → Patchμ t ty
_⊔μ_ da db with costL da ⩽?-ℕ costL db
...| yes _ = da
...| no  _ = db
```

As we shall see in section 2.3.1, this notion of cost is very delicate. The idea, however, is simple. If the heads are equal we have $d3 =$ Dμ-dwn (gdiff hdX hdX) $\cdots$, which is how copy is implemented. We want to copy as much as possible, so we will ensure that for all a, gdiff a a, that is, the identity patch for a, has cost 0 whereas Dμ-ins and Dμ-del will have strictly positive cost.

Since we mentioned *the identity patch* for an object, we might already introduce the two *special* patches that we need before talking about cost.

The Identity Patch Given the definition of gdiff, it is not hard to see that whenever $x \equiv y$, we produce a patch without any D-setl, D-setr, Dμ-ins or Dμ-del, let us call these the *change-introduction* constructors. Well, one can spare the comparisons made by gdiff and trivially define the identity patch for an object x, gdiff-id x, by induction on x. A good sanity check, however, is to prove that this intuition is in fact correct:

```
gdiff-id-correct
  : {n : ℕ}{t : Tel n}{ty : U n}
  → (a : El U ty t) → gdiff-id a ≡ gdiff a a
```

The Inverse Patch If a patch gdiff x y is not the identity, then it has *change-introduction* constructors. if we swap every D-setl for D-setr, and Dμ-ins for Dμ-del and vice-versa, we get a patch that transforms y into x. We shall call this operation the inverse of a patch.

```
D-inv : {n : ℕ}{t : Tel n}{ty : U n}
  → Patch t ty → Patch t ty
```

As one would expect, gdiff y x or D-inv (gdiff x y) should be the same patch. In fact, we have that gdiff y $x \approx$ D-inv (gdiff x y). That is to say gdiff y x behaves the same as D-inv (gdiff x y), but may not be identitcal. In the presence of equal cost alternatives they may make different choices.

2.3.1 The Cost Function

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

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

The question is, how should we do this? The cost of transforming x and y intuitively leads one to think about *how far is x from y*. We see that the cost of a patch should not be too different from the notion of distance.

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

In order to achieve a meaningful definition, we will impose the specification that our cost function must make the distance we defined above into a metric. We then proceed to calculate the cost function from its specification. Remember that we call a function *dist* a *metric* iff:

$$dist\ x\ y = 0 \quad \Leftrightarrow \quad x = y \tag{1}$$
$$dist\ x\ y \quad = \quad dist\ y\ x \tag{2}$$
$$dist\ x\ y + dist\ y\ z \quad \geqslant \quad dist\ x\ z \tag{3}$$

Equation (1) tells that the cost of not changing anything must be 0, therefore the cost of every non-*change-introduction* constructor should be 0. The identity patch then has cost 0 by construction, as we seen it is exactly the patch with no *change-introcution* constrcutor.

Equation (2), on the other hand, tells that it should not matter whether we go from x to y or from y to x, the effort is the same. In patch-space, this means that the inverse of a patch should preserve its cost. Well, the inverse operation leaves everything unchanged but flips the *change-introduction* constructors to their dual counterpart. We will hence assign a cost $c_\oplus = \mathsf{cost}\ \mathsf{D\text{-}setl} = \mathsf{cost}\ \mathsf{D\text{-}setr}$ and $c_\mu = \mathsf{cost}\ \mathsf{D}\mu\text{-}\mathsf{ins} = \mathsf{cost}\ \mathsf{D}\mu\text{-}\mathsf{del}$ and guarantee this by construction already. Some care must be taken however, as if we define c_μ and $c_\oplus$ as constants we will say that inserting a tiny object has the same cost of inserting a gigantic object! That is not what we are looking for in a fine-tuned diff algorithm. Let us then define $c_\oplus\ x\ y = \mathsf{cost}\ (\mathsf{D\text{-}setr}\ x\ y) = \mathsf{cost}\ (\mathsf{D\text{-}setl}\ x\ y)$ and $c_\mu\ x = \mathsf{cost}\ (\mathsf{D}\mu\text{-}\mathsf{ins}\ x) = \mathsf{cost}\ (\mathsf{D}\mu\text{-}\mathsf{del}\ x)$ so we can take this fine-tuning into account.

Equation (3) is concerned with composition of patches. The aggregate cost of changing x to y, and then y to z should be greater than or equal to changing x directly to z. We do not have a composition operation over patches yet, but we can see that this is trivially satisfied. Let us denote the number of *change-introduction* constructors in a patch p by $\#p$. In the best case scenario, $\#(\mathsf{gdiff}\ x\ y) + \#(\mathsf{gdiff}\ y\ z) = \#(\mathsf{gdiff}\ x\ z)$, this is the situation in which the changes of x to y and from y to z are non-overlapping. If they are overlapping, then some changes made from x to y must be changed again from y to z, yielding $\#(\mathsf{gdiff}\ x\ y) + \#(\mathsf{gdiff}\ y\ z) > \#(\mathsf{gdiff}\ x\ z)$.

From equations (1) and (2) and from our definition of the equality patch and the inverse of a patch we already have that:

```
gdiff-id-cost : {n : ℕ}{t : Tel n}{ty : U n}
  → (a : ElU ty t) → cost (gdiff-id a) ≡ 0

D-inv-cost : {n : ℕ}{t : Tel n}{ty : U n}
  → (d : Patch t ty)
  → cost d ≡ cost (D-inv d)
```

In order to finalize our definition, we just need to find an actual value for $c_\oplus$ and c_μ. Note that we have a lot of freedom to choose these values. And in fact, they are what is going to drive the diff algorithm to prefer some changes over others. Let us then take a look at where the difference between these values comes into play, and calculate from there. Assume we have stopped execution of gdiffL at the $d_1 \sqcup_\mu d_2 \sqcup_\mu d_3$ expression. Here we have three patches to choose from:

$$\begin{array}{lcl} d_1 & = & \mathsf{D}\mu\text{-}\mathsf{ins}\ hdY\ ::\ \mathsf{gdiffL}\ (x\ ::\ xs)\ (chY\ +\!\!+\ ys) \\ d_2 & = & \mathsf{D}\mu\text{-}\mathsf{del}\ hdX\ ::\ \mathsf{gdiffL}\ (chX\ +\!\!+\ xs)\ (y\ ::\ ys) \\ d_3 & = & \mathsf{D}\mu\text{-}\mathsf{dwn}\ (\mathsf{gdiff}\ hdX\ hdY) \\ & :: & \mathsf{gdiffL}\ (chX\ +\!\!+\ xs)\ (chY\ +\!\!+\ ys) \end{array}$$

We will only compare d_1 and d_3, as the cost of inserting and deleting should be the same, the analisys for d_2 is analogous. By choosing d_1, we would be opting to insert hdY instead of transforming hdX into hdY, this is preferable only when we do not have to delete hdX later on, in $\mathsf{gdiffL}\ (x\ ::\ xs)\ (chY\ +\!\!+\ ys)$, as that would be a waste of information. Deleting hdX is inevitable when $hdX \notin chY\ +\!\!+\ ys$. Assuming without loss of generality that this deletion happens in the next step, we have:

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

Hence, $\mathsf{cost}\ d_1$ is $c_\mu\ \ hdX + c_\mu\ \ hdY + w$, for $w = \mathsf{cost}\ (\mathsf{tail}\ d_3)$. Obviously, hdX and hdY are values of the same type. Namelly $hdX, hdY : \mathsf{ElU}\ ty\ (\mathsf{tcons}\ \mathsf{u1}\ t)$. Since we want to apply this to Haskell datatypes by the end of the day, it is acceptable to assume that ty is a coproduct of constructors. Hence hdX and hdY are values of the same finitary coproduct, representing the constructors of the fixed-point datatype. If hdX and hdY comes from different constructors of our fixed-point datatype in question, then[2] $hdX = i_j\ x'$ and $hdY = i_k\ y'$ where $j \neq k$. The patch from hdX to hdY will therefore involve a $\mathsf{D\text{-}setl}\ x'\ y'$ or a $\mathsf{D\text{-}setr}\ y'\ x'$, hence the cost of d_3 becomes $c_\oplus\ x'\ y' + w$. Remember that in this situation it is wise to delete and insert instead of recursively changing. Since things are comming from a different constructors the structure of the outermost type is definetely changing, we want to reflect that! This means we need to select d_1 instead of d_3, hence:

$$\begin{array}{rcl} c_\mu\ (i_j\ x') + c_\mu\ (i_k\ y') + w & < & c_\oplus\ (i_j\ x')\ (i_k\ y') + w \\ \Leftrightarrow \quad c_\mu\ (i_j\ x') + c_\mu\ (i_k\ y') & < & c_\oplus\ (i_j\ x')\ (i_k\ y') \end{array}$$

If hdX and hdY come from the same constructor, on the other hand, the story is slightly different. We now have $hdX = i_j\ x'$ and $hdY = i_j\ y'$, the cost of d_1 still is $c_\mu\ (i_j\ x') + c_\mu\ (i_k\ y') + w$ but the cost of d_3 is $\mathsf{dist}\ x'\ y' + w$, since $\mathsf{gdiff}\ hdX\ hdY$ will be $\mathsf{gdiff}\ x'\ y'$ preceded by a sequence of D-inr and D-inr since hdX and hdY they come from the same coproduct injection, but these have cost 0. This is the situation that selecting d_3 might be a wise choice, therefore we need:

$$\begin{array}{rcl} \mathsf{dist}\ x'\ y' + w & < & c_\mu\ (i_j\ x') + c_\mu\ (i_k\ y') + w \\ \Leftrightarrow \quad \mathsf{dist}\ x'\ y' & < & c_\mu\ (i_j\ x') + c_\mu\ (i_k\ y') \end{array}$$

In order to enforce this behaviour on our diffing algorithm, we need to assign values to c_μ and $c_\oplus$ that respects:

$$\mathsf{dist}\ x'\ y' < c_\mu\ (i_j\ x') + c_\mu\ (i_k\ y') < c_\oplus\ (i_j\ x')\ (i_k\ y')$$

[2] We use i_j to denote the j-th injection into a finitary coproduct.

Note that there are an infinite amount of definitions that would fit here. This is indeed a central topic of future work, as the cost function is what drives the diffing algorithm. So far we have calculated a relation between c_μ and $c_\oplus$ that will make the diff algorithm match in a top-down manner. That is, the outermost type is seen as the heaviest change. Different domains may require a different relation. For a first generic implementation, however, this relation does make sense. Now is the time for assigning values to c_μ and $c_\oplus$ and the diffing algorithm is completed. We simply define the cost function in such a way that it has to satisfy the constraints we imposed.

```
cost (D-A ())
cost  D-void    = 0
cost (D-inl d)  = cost d
cost (D-inr d)  = cost d
cost (D-setl xa xb) = 2 * (sizeElU xa + sizeElU xb)
cost (D-setr xa xb) = 2 * (sizeElU xa + sizeElU xb)
cost (D-pair da db) = cost da + cost db
cost (D-β d)    = cost d
cost (D-top d) = cost d
cost (D-pop d) = cost d
cost (D-mu l)  = costL l

costμ : {n : ℕ}{t : Tel n}{ty : U (suc n)}
      → Dμ ⊥ₚ t ty → ℕ
costμ (Dμ-A ())
costμ (Dμ-ins x) = 1 + sizeElU x
costμ (Dμ-del x) = 1 + sizeElU x
costμ (Dμ-dwn x) = cost x
```

Where costL $= sum \cdot map$ costμ and the size of an element is defined by:

```
sizeElU : {n : ℕ}{t : Tel n}{u : U n} → ElU u t → ℕ
sizeElU void       = 1
sizeElU (inl el)   = 1 + sizeElU el
sizeElU (inr el)   = 1 + sizeElU el
sizeElU (ela , elb) = sizeElU ela + sizeElU elb
sizeElU (top el)  = sizeElU el
sizeElU (pop el)  = sizeElU el
sizeElU (mu el)
  = let (hdE , chE) = μ-open (mu el)
    in sizeElU hdE + foldr _+_ 0 (map sizeElU chE)
sizeElU (red el)  = sizeElU el
```

2.4 A Small Example

- *add context*

I) Copy the first line;

II) Enter the second line;

  i) Copy the first field;

  ii) Enter the second field;

    - Update atom `"1"` for atom `"2"`.

  iii) Copy the third field;

III) Copy the third line.

IV) Finish.

In figure 1 we show the patch that corresponds to that.

Consider now Bob's structural changes to the CSV file[3]. If you overlap both, you should notice that there is *Upd* operation on top of another. This was in fact expected given that Alice and Bob performed changes in disjoint parts of the CSV file.

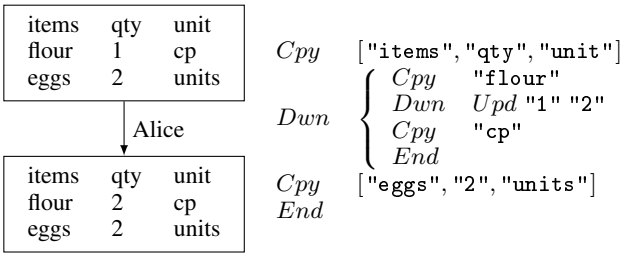

Figure 1. Alice's Patch

2.5 Applying Patches

At this stage we are able to: work generically on a suitable universe; describe how elements of this universe can change and compute those changes. In order to make our framework useful, though, we need to be able to apply the patches we compute. The application of patches is easy, for we will only show the implementation for coproducts and fixedpoints here. The rest is very straightforward.

```
gapply : {n : ℕ}{t : Tel n}{ty : U n}
  → Patch t ty → ElU ty t → Maybe (ElU ty t)
gapply (D-inl diff) (inl el) = inl <M> gapply diff el
gapply (D-inr diff) (inr el) = inr <M> gapply diff el
gapply (D-setl x y) (inl el) with x ≟-U el
... | yes _ = just (inr y)
... | no  _ = nothing
gapply (D-setr y x) (inr el) with y ≟-U el
... | yes _ = just (inl x)
... | no  _ = nothing
gapply (D-setr _ _) (inl _) = nothing
gapply (D-setl _ _) (inr _) = nothing
gapply (D-inl diff) (inr el) = nothing
gapply (D-inr diff) (inl el) = nothing
gapply {ty = μ ty} (D-mu d) el = gapplyL d (el :: []) >>= safeHead
  ⋮

gapplyL : {n : ℕ}{t : Tel n}{ty : U (suc n)}
  → Patchμ t ty → List (ElU (μ ty) t) → Maybe (List (ElU (μ ty) t))
gapplyL [] [] = just []
gapplyL [] _  = nothing
gapplyL (Dμ-A () :: _)
gapplyL (Dμ-ins x  :: d) l = gapplyL d l >>= glns x
gapplyL (Dμ-del x  :: d) l = gDel x l  >>= gapplyL d
gapplyL (Dμ-dwn dx :: d) [] = nothing
gapplyL (Dμ-dwn dx :: d) (y :: l) with μ-open y
... | hdY , chY with gapply dx hdY
... | nothing = nothing
... | just y' = gapplyL d (chY ++ l) >>= glns y'
```

Where <M> is the applicative-style application for the *Maybe* monad; >>= is the usual bind for the *Maybe* monad and safeHead is the partial head function with type $[a] \to Maybe\ a$. In gapplyL, we have a glns function, which will get a head and a list of children of a fixed point, will try to μ-close it and add the result to the head of the remaining list. On the other hand, gDel will μ-open the first element of the received list, compare it with the current head and return the tail of the input list appended to its children.

Instead of presenting an example, lets provide some intuition for our gapply function. Looking at the D-setl case, for instance.

[3] Exercise to the reader! Clue: the last two operations are *Ins* `["sugar","1","tsp"]` *End*

gapply (D-set⌊ x y) is expecting to transform a in⌊ x into a inr y. Upon receiving a in⌊ value, we need to check whether or not its contents are equal to x. If they are, we are good to go. If not, we have to return nothing as we cannot possibly know what to do. If we look instead on the D-in⌊ $diff$ branch, we see that it only succeeds upon receiving a in⌊ x, given that gapply $diff$ succeeds in modifying x.

The important part of application, nevertheless, is that it must produce the expected result. A correctness result guarantees that. Its proof is too big to be shown here but it has type:

```
correctness : {n : ℕ}{t : Tel n}{ty : U n}
  → (a b : El⌊U ty t)
  → gapply (gdiff a b) a ≡ just b
```

Also relevant is the fact that the inverse of a patch behaves as it should:

```
D-inv-sound
  : {n : ℕ}{t : Tel n}{ty : U n}
  → (a b : El⌊U ty t)
  → gapply (D-inv (gdiff a b)) b ≡ just a
```

We have given algorithms for computing and applying differences over elements of a generic datatype. Moreover, we proved our algorithms are correct with respect to each other. This functionality is necessary for constructing a version control system, but it is by no means sufficient!

3. Patch Propagation

In a nutshell, any version control system must accomplish two tasks: (A) we need to be able to produce and apply patches and (B) we need to be able to merge different, concurrent, changes make to the same object. We have taken care of task (A) in the previous sections, and even though current VCS tools already excel at it, there is a big lack of tools exceling at (B). All the structural information we are using in task (A) is, in fact, providing a lot more to help us at task (B), as we shall discuss in this section.

The task of merging changes arise when we have multiple users changing the same file at the same time. Imagine Bob and Alice perform concurrent edits in an object A_0, which are captured by patches p and q. The center of the repository needs to keep only one copy of that object, but upon receiving the changeset of both Bob and Alice we have:

$$A_1 \xleftarrow{p} A_0 \xrightarrow{q} A_2$$

Our idea is to imcorporate the changes expressed by p into a new patch, namelly one that is aimed at being applied somewhere already changed by q, and vice-versa, in such a way that they converge. We call this the residual patch. The diagram in figure 2 illustrates the result of merging p and q through propagation.

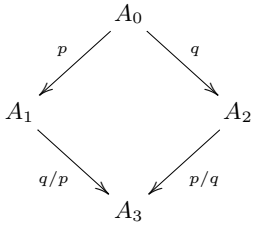

Figure 2. Residual patch square

The residual p/q of two patches p and q captures the notion of incorporating the changes made by p in an object that has already been modified by q.

In an ideal world, we would expect the residual function to have type Patch t ty → Patch t ty → Patch t ty. Real life is more complicated. To begin with, it only makes sense to compute the residual of patches that are *aligned*, that is, they can be applied to the same input. For this, we make the residual function partial though the $Maybe$ monad: Patch t ty → Patch t ty → Maybe (Patch t ty) and define two patches to be aligned if and only if their residual returns a just.

Partiality solves just a few problems, what if, for instance, Bob and Alice changes the same cell in their CSV file? Then it is obvious that someone (human) have to chose which value to use in the final, merged, version.

For this illustration, we will consider the conflicts that can arise from propagating the changes Alice made over the changes already made by Bob, that is, p_{alice}/p_{bob}.

- If Alice changes a_1 to a_2 and Bob changed a_1 to a_3, with $a_2 \neq a_3$, we have an *update-update* conflict;
- If Alice deletes information that was changed by Bob we have an *delete-update* conflict;
- Last but not least, if Alice changes information that was deleted by Bob we have an *update-delete* conflict.
- If Alice adds information to a fixed-point, this is a *grow-left* conflict;
- When Bob added information to a fixed-point, which Alice didn't, a *grow-right* conflict arises;
- If both Alice and Bob add different information to a fixed-point, a *grow-left-right* conflict arises;

Most of the readers might be familiar with the *update-update*, *delete-update* and *update-delete* conflicts, as these are the most straight forward to be recognized as conflicts. We refer to these conflicts as *update* conflicts.

The *grow* conflicts are slightly more subtle. This class of conflicts corresponds to the *alignment table* that *diff3* calculates [**?**] before deciding which changes go where. The idea is that if Bob adds new information to a file, it is impossible that Alice changed it in any way, as it was not in the file when Alice was eiditing it, we then flag it as a conflict. The *grow-left* and *grow-right* are easy to handle, if the context allows, we could simply transform them into atual insertions or copies. They represent insertions made by Bob and Alice in *disjoint* places of the structure. A *grow-left-right* is more complex, as it corresponds to a overlap and we can not know for sure which should come first unless more information is provided. From the structure in our patch-space, we can already separate conflicts by the types they can occur on. An *update-update* conflict has to happen on a coproduct type, whereas the rest are restricted to fixed-point types. In Agda,

```
data C : {n : ℕ} → Tel n → U n → Set where
  UpdUpd : {n : ℕ}{t : Tel n}{a b : U n}
         → ElU (a ⊕ b) t → ElU (a ⊕ b) t → ElU (a ⊕ b) t
         → C t (a ⊕ b)
  DelUpd : {n : ℕ}{t : Tel n}{a : U (suc n)}
         → ValU a t → ValU a t → C t (μ a)
  UpdDel : {n : ℕ}{t : Tel n}{a : U (suc n)}
         → ValU a t → ValU a t → C t (μ a)
  GrowL  : {n : ℕ}{t : Tel n}{a : U (suc n)}
         → ValU a t → C t (μ a)
  GrowLR : {n : ℕ}{t : Tel n}{a : U (suc n)}
         → ValU a t → ValU a t → C t (μ a)
  GrowR  : {n : ℕ}{t : Tel n}{a : U (suc n)}
         → ValU a t → C t (μ a)
```

- *Pijul has this notion of handling a merge as a pushout, but it uses the free co-completion of a rather simple category. This doesn't give enough information for structured conflict solving.*
- *BACK THIS UP!*

3.1 Incorporating Conflicts

In order to track down these conflicts we need a more expressive patch data structure. We exploit D's parameter for that matter. This approach has the advantage of separating conflicting from conflict-free patches on the type level, guaranteing that we can only *apply* conflict-free patches.

The final type of our residual[4]. operation is:

```
_/_ : {n : ℕ}{t : Tel n}{ty : U n}
  → Patch t ty → Patch t ty → Maybe (D C t ty)
```

We reiterate that the partiality comes from the fact the residual is not defined for non-aligned patches. The whole function is too big to be shown here, but explaining one of its cases can provide valuable intuition.

```
res (Dμ-dwn dx :: dp) (Dμ-del y :: dq)
  with gapply dx y
...| nothing = nothing
...| just y' with y ≟-U y'
...| yes _ = res dp dq
...| no  _ = _ :: _ (Dμ-A (UpdDel y' y)) <M> res dp dq
```

Here we are computing the residual:

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

We want to describe how to apply the P_x changes to an object that has been modified by the P_y patch. Note the order is important! The first thing we do is to check whether or not the patch dx can be applied to y. If we can not apply dx to y, then patches P_x and P_y are non-aligned, we then simply return nothing. If we can apply dx to y, however, this will result in an object y'. We then need to compare y to y', as if they are different we are in a UpdDel conflict situation. If they are equal, then dx is just diff-id y, that is, no changes were performed. To extend this to be applied to the object were y was deleted we simply suppress the Dμ-del and continue recursively. The remaining cases follow a similar reasoning process.

The attentive reader might have noticed a symmetric structure on conflicts. This is not at all by chance. In fact, we can prove that the residual of p/q have the same (modulo symmetry) conflicts as q/p. This proof goes in two steps. Firstly, residual-symmetry proves that the symmetric of the conflicts of p/q appear in q/p, but this happens modulo a function. We then prove that this function does not introduce any new conflicts, it is purely structural.

```
residual-symmetry-thm
  : {n : ℕ}{t : Tel n}{ty : U n}{k : D C t ty}
  → (d1 d2 : Patch t ty)
  → d1 / d2 ≡ just k
  → Σ (D C t ty → D C t ty)
      (λ op → d2 / d1 ≡ just (D-map C-sym (op k)))

residual-sym-stable : {n : ℕ}{t : Tel n}{ty : U n}{k : D C t ty}
  → (d1 d2 : Patch t ty)
  → d1 / d2 ≡ just k
  → forget <M> (d2 / d1) ≡ just (map (↓-map-↓ C-sym) (forget k))
```

Here $\downarrow\!-map-\!\downarrow$ takes care of hiding type indexes and forget is the cannonical function with type D A t ty → List ($\downarrow$ A), $\downarrow\_$ encapsulates the type indexes of the different A's we might come accross.

Now, we can compute both p/q and q/p at the same time. It also backs up the intuition that using residuals or patch commutation (as in darcs) is not significantly different.

This means that p/q and q/p, although different, have the same conflicts (up to symmetry). Hence, we can (informally) define a *merge strategy* as a function $m : \forall\{t\ ty\} \to \mathsf{C}\ t\ ty \to \mathsf{D}\ \bot_p\ t\ ty$, which can now be mapped over D C t ty pointwise on its conflicts. We end up with an object of type D (D $\bot_p$) t ty. This is not a problem, however, since the free-monad structure on D provides us with a multiplication μ_D : D (D A) t ty → D A t ty. Therefore, $merge_1\ m$: D C t ty $\xrightarrow{\mu_D \cdot D\ m}$ Patch t ty is one possible *merge tool* for removing the conflicts of a patch. Mapping m over the conflicting patch is by far not the only possible way of walking the tree, as we shall see in section 4. This opens up a lot of very interesting questions and paves the road to defining conflict resolution combinators. Allowing for a great degree of genericity in the base framework.

4. A Haskell Prototype

In sections 2.1 and 3 we have layed the foundations for creating a functional version control system. We proceed by illustrating these with a prototype in Haskell, with an emphasis on its extended capability of handling conflicts.

The user has access to a typeclass $Diffable\ a$, which gives the basic diffing and merging functionality for objects of type a:

```
class (Sized a) ⇒ Diffable a where
  diff  :: a → a → Patch a
  apply :: Patch a → a → Maybe a
  res   :: Patch a → Patch a → Maybe (PatchC a)
  cost  :: Patch a → Int
```

Where $Sized\ a$ is a class providing the sizeElU function, presented in section 2.3.1; $Patch$ is a GADT[**?**] reflecting our Patch type in Agda. We then proceed to provide instances by induction on the structure of a. Products and coproducs are trivial and follow immediatly from the Agda code.

```
instance (Diffable a, Diffable b)
  ⇒ Diffable (a, b)
instance (Eq a, Eq b, Diffable a, Diffable b)
  ⇒ Diffable (Either a b)
```

Fixed points are not complicated, too. It is important that they support the same plugging and unplugging functionality as in Agda, though. We have to use explicit recursion since current

[4] Our residual operation does not form a residual as in the Term Rewriting System sense[**?**]. It might, however, satisfy interesting properties. This is left as future work for now

Haskell's instance search does not have explicit type applications yet.

```
newtype Fix a = Fix {unFix :: a (Fix a)}
class (Eq (a ())) ⇒ HasAlg (a :: ∗ → ∗) where
  ar    :: Fix a → Int
  ar    = length ∘ ch
  ch    :: Fix a → [Fix a]
  hd    :: Fix a → a ()
  close :: (a (), [Fix a])
        → Maybe (Fix a, [Fix a])
instance (HasAlg (a :: ∗ → ∗), Diffable (a ()))
  ⇒ Diffable (Fix a)
```

All the other types can also be seen as sums-of-products. We then define a class and some template Haskell functionality to generate instances of $SOP\ a$. The *overlappable* pragma makes sure that Haskell's instance search will give preference to the other instances.

```
class HasSOP (a :: ∗) where
  type SOP a :: ∗
  go :: a → SOP a
  og :: SOP a → a
instance {-# OVERLAPPABLE #-}
  (HasSOP a, Diffable (SOP a))
  ⇒ Diffable a
```

As the tool is still a very young prototype, we chose to ommit implementation details such as memoization or explicit

4.1 A more involved example

- *Show how refactoring works*

5. Sketching a Control Version System

- Different views over the same datatype will give different diffs.
- **newtype** annotations can provide a gread bunch of control over the algorithm.
- Directories are just rosetrees...

6. Summary, Remarks and Related Work

To Research!

- *Check out the antidiagonal with more attention:* `http://blog.sigfpe.com/2007/09/type-of-distinct-pairs.html`
  - ***ANS:*** *Diffing and Antidiagonals are fundamentally different. The antidiagonal for a type T is a type X such that there exists $X \to T^2$. That is, X produces two **distinct** T's, whereas a diff produces a T given another T!*
- *related work goes here!*
- *Add some boilerplate to the shameful part of the paper...*

6.1 Cost, Inverses and Lattices

Back in section 2.3.1, where we calculated our cost function from a specification, we did not provide a formal proof that our definitions did in fact satisfy the relation we stated:

$$\mathsf{dist}\ x'\ y' < c_\mu\ (i_j\ x') + c_\mu\ (i_k\ y') < c_\oplus\ (i_j\ x')\ (i_k\ y')$$

This is, in fact, deceptively complicated. Since $\_\sqcup_\mu\_$ is not commutative nor associative, in general, we do not have much room to reason about the result of a gdiffL. A more careful definition of $\_\sqcup_\mu\_$ that can provide more properties is paramount for a more careful study of the cost function. Defining $\_\sqcup_\mu\_$ in such a way that it gives us a lattice over patches and still respects patch inverses is very tricky, as for making it preserve inverses we need to have $\mathsf{D\text{-}inv}(d_1 \sqcup_\mu d_2) \equiv \mathsf{D\text{-}inv}\ d_1 \sqcup_\mu \mathsf{D\text{-}inv}\ d_2$. A simpler option would be to aim for a quasi-metric d instead of a metric (dropping symmetry; equation (2)), this way inverses need not to distribute over $\_\sqcup_\mu\_$ and we can still define a metric q, but now with codomain $\mathbb{Q}$, as $q\ x\ y = \frac{1}{2}(d\ x\ y - d\ y\ x)$.

6.2 A remark on Type Safety

The main objectives of this project is to release a solid diffing and merging tool, that can provide formal guarantees, written in Haskell. The universe of user-defined Haskell types is smaller than context free types; in fact, we have fixed-points of sums-of-products. Therefore, we should be able to apply the knowledge acquired in Agda directly in Haskell. In fact, we did so! With a few adaptations here and there, to make the type-checker happy, the Haskell code is almost a direct translation. There is one minor detail we would like to point out in our approach so far. Our $\mathsf{D}\mu$ type admits ill-formed patches. Consider the following simple example:

```
nat : U 0
nat = μ (u1 ⊕ vl)

ill-patch : Patch tnil nat
ill-patch = D-mu (Dμ-ins (inr (top void)) :: [])

prf : ∀ el → gapply ill-patch el ≡ nothing
prf el = refl
```

On ill-patch we try to insert a suc constructor, which require one recursive argument, but then we simply end the patch. Agda realizes that this patch will lead nowhere in an instant.

Making $\mathsf{D}\mu$ type-safe by construction should be simple. The type of the functor is always fixed, the telescope too. These can become parameters. We just need to add two $\mathbb{N}$.

```
data Dμ {a}(A : {n : ℕ} → Tel n → U n → Set a)
   {n : ℕ}(t : Tel n)(ty : U (suc n))
: ℕ → ℕ → Set where
```

Then, $\mathsf{D}\mu\ A\ t\ ty\ i\ j$ will model a patch over elements of type $T = \mathsf{El}\mathsf{U}(\mu\ ty)\ t$ and moreover, it expects a vector of length i of T's and produces a vector of length j of T's. This is very similar to how type-safety is guaranteed in [**?**], but since we have the types fixed, we just need the arity of their elements.

If we try to encode this in Agda, using the universe of context-free types, we run into a very subtle problem. In short, we can not prove that if two elements come from the same constructor, they have the same arity. This hinders $\mathsf{D}\mu$-dwn useless. To fix this, we need to add kinds to our universe. Context-free type aplication does not behave the same way as in Haskell. Consider list as defined in section 2.1 and ltree as defined below.

```
ltree : {n : ℕ} → U (2 + n)
ltree = μ (wk (wk vl) ⊕ wk vl ⊗ vl ⊗ vl)
```

The Haskell equivalent, with explicit recursion, would be:

```
data LTreeF a b x = Leaf a | Fork b x x
type LTree a b    = Fix (LTreeF a b)
```

The kind of $LTree$ is $\star \Rightarrow \star \Rightarrow \star$. Hence, it can only receive arguments of kind $\star$. In CF, however, we can apply ltree to list:

$\mathcal{M} : \{n : \mathbb{N}\} \to \mathsf{U}\ (\mathsf{suc}\ n)$
$\mathcal{M} = \beta\ |\mathsf{tree}\ |\mathsf{list}$

In Haskell, this type would be defined as $\mathbf{type}\ \mathcal{M}\ a = \mathit{LTree}\ a\ [a]$. The existence of these non-intuitive types makes it extremely complicated, if not impossible, to formally state properties relating the arity of an element and the arity of its type, moreover, CFdoes not differentiate between sums-of-products.

Fixing the type-unsafety we have in our model is not very difficult. By encoding our algorithm in a kinded universe that more closely resemble Haskell types does the trick. This is left as future work.

7. Conclusion

- This is what we take out of it.