Best Title in the Universe

42

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

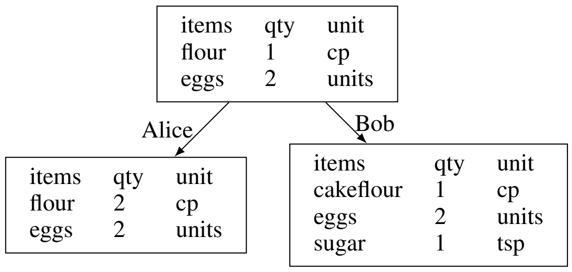
General Terms Haskell

Keywords Haskell

1. Introduction

The majority of version control systems handle patches in a non-structured way. They see a file as a list of lines that can be inserted, deleted or modified, with no regard to the semantics of that specific file. The immediate consequence of such design decision is that we, humans, have to solve a large number of conflicts that arise from, in fact, non conflicting edits. Implementing a tool that knows the semantics of any file we happen to need, however, is no simple task, specially given the plethora of file formats we see nowadays.

This can be seen from a simple example. Lets imagine Alice and Bob are iterating over a cake's recipe. They decide to use a version control system and an online repository to keep track of their modifications.



items	qty	unit
flour	1	cp
eggs	2	units

items	qty	unit
flour	2	cp
eggs	2	units

items	qty	unit
cakeflour	1	cp
eggs	2	units
sugar	1	tsp

Figure 1. Sample CSV files

Lets say that both Bob and Alice are happy with their independent changes and want to make a final recipe. The standard way to track differences between files is the `diff3` [FIXBIB] unis tool. Running `diff3 Alice.csv O.csv Bob.csv` would result in the output presented in figure 2. Every tag `====` marks a difference. Three locations follows, formatted as `file:line type`. The change type can be a *Change*, *Append* or *Delete*. The first one, says that file 1 (`Alice.csv`) has a change in line 2 (`1:2c`) which is `flour, 2 , cp`; and files 2 and 3 have different changes in the same line. The tag `====3` indicates that there is a difference in file 3 only. Files 1 and 2 should append what changed in file 3 (line 4).

```
====
1:2c
  flour, 2  , cp
2:2c
  flour, 1  , cp
3:2c
  cakeflour, 1  , cp
====3
1:3a
2:3a
3:4c
  sugar, 1  , tsp
```

Figure 2. Output from `diff3`

If we try to merge the changes, `diff3` will flag a conflict and therefore require human interaction to solve it, as we can see by the presence of the `====` indicator in its output. However, Alice's and Bob's edits, in figure 1 do *not* conflict, if we take into account the semantics of CSV files. Although there is an overlapping edit at line 1, the fundamental editing unit is the cell, not the line.

We propose a structural diff that is not only generic but also able to track changes in a way that the user has the freedom to decide which is the fundamental editing unit. Our work was inspired by [**?**] and [**?**]. We did extensive changes in order to handle structural merging of patches. We also propose extensions to this algorithm capable of detecting purely structural operations such as refactorings and cloning.

The paper begins by exploring the problem, generically, in the Agda [FIXBIB] language. Once we have a provably correct algorithm, the details of a Haskell implementation of generic diff'ing are sketched. To open ground for future work, we present a few extensions to our initial algorithm that could be able to detect semantical operations such as *cloning* and *swapping*.

Contributions

- *Study of a more algebraic patch theory.*
- *Agda model.*
- *Haskell Prototype.*

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

Alice and Bob were both editing a CSV file which represents data that is isomorphic to $[[Atom\ String]]$, where $Atom\ a$ is a simple tag that indicates that as should be treated abstractly, that is, either they are equal or different, we will not open these values to check for structural changes.

As we are tracking differences, there are a few operations that are inherent to our domain, such as: inserting; deleting; copying and updating. When we say *structural diffing*, however, we add another option to this list. Now we will also be able to go down the structure of some object and inspect its parts. To illustrate this, let us take Alice's change as in figure 1, her changes to the file could be described, structurally, as:

I) Copy the first line;

II) Enter the second line;

  i) Copy the first field;

  ii) Enter the second field;

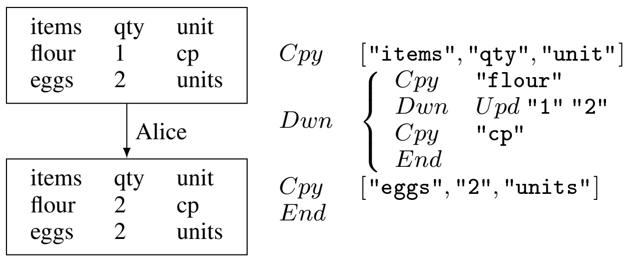
   - Update atom "1" for atom "2".

  iii) Copy the third field;

III) Copy the third line.

IV) Finish.

In figure 3 we show the patch that corresponds to that.



items	qty	unit
flour	1	cp
eggs	2	units

$\downarrow$ Alice

items	qty	unit
flour	2	cp
eggs	2	units

```
Cpy   ["items","qty","unit"]
Dwn   { Cpy   "flour"
      { Dwn   Upd "1" "2"
      { Cpy   "cp"
      { End
Cpy   ["eggs","2","units"]
End
```

Figure 3. Alice's Patch

Consider now Bob's structural changes to the CSV file[1]. 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.

- *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*

- *connect this section and the next*

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

2.1 Context Free Datatypes

Although our running example, of CSV files, has type $[[Atom\ String]]$, lists of a themselfes are in fact the least fixed point of the functor $X \mapsto 1 + a \times X$. Which is a *context-free type*, in the sense of [**?**]. For it is constructed following the grammar CF of context free types with a deBruijn representation for variables.

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

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)
```

Here, β stands for type application; vl is the topmost variable in scope and wk ignores the topmost variable in scope. 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.

We stress that one of 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, and will be discussed in section 6.

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. Long story short, we have to use a decreasing Telescope to satisy the termination checker. In Agda, the elements of U are defined by:

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

The Tel index is the telescope in which to look for the instantiation of type-variables. A value $(v\ :\ \mathsf{ElU}\ \{n\}\ ty\ t)$ reads roughly as: a value of type ty with n variables, applied to n types t with at most $n-1$ variables. We need this decrease of type variables to convince the termination checker that our code is ok. It's Agda definition is:

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

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} → ElU list (tcons u1 t)
myList = mu (inr (pop (top void) , top (mu (inl void))))
```

So far so good. We seem to have the syntax figured out. But which operations can we perform to 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 : ElU 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. Things like getting the list of immediate children, or computing their arity, that is, how many children do they have, are very usefull.

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

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

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

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

- *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 A 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[2]. 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.

For T being the Unit type, we can not modify it.

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

For T being 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)
```

For T being 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)
```

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 4.1, 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] Types that do not have any free type-variables

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 looks like:

```
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.

Fixed Points have a fundamental difference over regular algebraic datatypes. They can grow or shrink arbitralily. 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
```

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
```

But the interesting bits are the *edit operations* we allow, where $\mathsf{Val}\ a\ t = \mathsf{ElU}\ a\ (\mathsf{tcons}\ \mathsf{u1}\ t)$:

```
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-cpy : {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 t (β a u1) → Dμ A t a
```

The reader familiar with [**?**] will notice that they are almost the same (adapted to our choice of universe), with two differences: we admit a new constructur, Dμ-dwn; and our diff type is less type-safe. The type-safety concerns will be discussed in section 5.2.

Before we delve into diffing fixed poitn values, we show some specialization of our generic operations to fixed points. Given that $\mu X.F\ \ X\ \approx\ F\ \ 1 \times [\mu X.F\ \ X]$, 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 are more convenient:

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


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


μ-close : {n : ℕ}{t : Tel n}{ty : U (suc n)}
  → Openμ t ty → Maybe (ElU (μ ty) t × List (ElU (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. The following soundness lemma guarantees the correct behaviour;

```
μ-close-resp-arity
  : {n : ℕ}{t : Tel n}{ty : U (suc n)}{a : ElU (μ ty) t}
    {hdA : ElU ty (tcons u1 t)}{chA l : List (ElU (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 (ElU (μ ty) t) → List (ElU (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) with μ-open x | μ-open y
...| hdX , chX | hdY , chY with hdX ≟-U hdY
...| no  _ = let
     d1 = Dμ-ins hdY :: (gdiffL (x :: xs) (chY ++ ys))
     d2 = Dμ-del hdX :: (gdiffL (chX ++ xs) (y :: ys))
     d3 = Dμ-dwn (gdiff (red hdX) (red hdY))
          :: (gdiffL (chX ++ xs) (chY ++ ys))
in d1 ⊔μ d2 ⊔μ d3
...| yes _ = let
     d3 = Dμ-cpy hdX :: (gdiffL (chX ++ xs) (chY ++ ys))
in 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 $y{:}ys$. The first thing we check is whether the heads are equal, if so, we force the copying. If they are not equal, we have three possible diffs that perform the required transformation. We then choose the one with *minimum cost*, in fact, _⊔μ_ will return the patch with the least cost. This cost notion is very delicate, for it will be discussed later, in section 2.5.

In fact, the example provided in figure 3 is a diff produced by our algorithm, with the constructors simplified to improve readability.

2.4 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 usefull, though, we need to be able to apply the patches we compute. To our luck, the application of patches is easy, for we will only show the implementation for coproducts and fixedpoints here. The rest is very straight forward.

```
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 »= gIns x
gapplyL (Dμ-del x  :: d) l = gDel x l  »= gapplyL d
gapplyL (Dμ-cpy x  :: d) l = gDel x l  »= gapplyL d »= gIns x
gapplyL (Dμ-dwn dx :: d) [] = nothing
gapplyL (Dμ-dwn dx :: d) (y :: l) with μ-open y
...| hdY , chY with gapply dx (red hdY)
...| nothing  = nothing
...| just (red y') = gapplyL d (chY ++ l) »= gIns y'
```

Where $<M>$ is the applicative-style application for the *Maybe* monad; $\gg=$ 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 gIns 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 $\mu{-}open$ 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.

The important part of application is that it must produce the expected result. Our correctness result, below, guarantees that. Its proof is too big to be shown here, however.

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

2.5 The Cost Function

- *the cost function should satisfy a few properties, such as: if x and y come from the same constructor, then $cost(diff\,xy) \leq cost(Delx :: Addy :: End)$. Otherwise, $gdiffL$ will always choose DmuDwn first.*

3. A Category of Patches

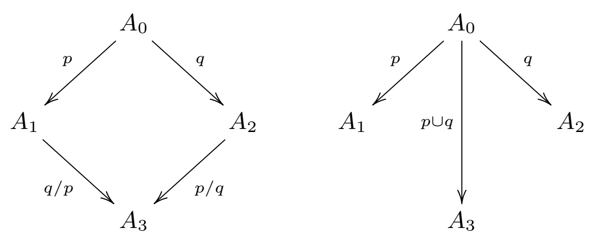
To Research!

- *Define patch composition, prove it makes a category.*
- *But then... does it make sense to compute the composition of patches?*
- *In a vcs setting, we always have the intermediate files that originated the patches, meaning that composition can be defined semantically by: $apply(p \cdot q) \equiv applyq \circ applyp$, where $\circ$ is the Kleisli composition of $+1$.*
- *This gives me an immediate category... how usefull is it?*

4. Patch Propagation

Let's say Bob and Alice perform edits in a given object, which are captured by patches p and q. In the version control setting, the natural question to ask is *how do we join these changes*.

There are two solutions that could possibly arise from this question. Either we group the changes made by p and by q (as long as they are compatible) and create a new patch to be applied on the source object, or, we calculate how to propagate the changes of p over q and vice-versa. Figure 4 illustrates these two options.



$$\begin{tikzcd} & A_0 \arrow[dl, "p"'] \arrow[dr, "q"] & \\ A_1 \arrow[dr, "q/p"'] & & A_2 \arrow[dl, "p/q"] \\ & A_3 & \end{tikzcd} \qquad \begin{tikzcd} & A_0 \arrow[dl, "p"'] \arrow[dr, "q"] \arrow[dd, "p\cup q"'] & \\ A_1 & & A_2 \\ & A_3 & \end{tikzcd}$$

Figure 4. Residual Square on the left; three-way-merging on the right

The residual p/q of two patches p and q only makes sense if both p and q are aligned, that is, are defined for the same input. It captures the notion of incorporating the changes made by p in an object that has already been modified by q.

We chose to use the residual notion, as it seems to have more structure into it. Not to mention we could define $p \cup q \equiv (q\ p) \cdot p \equiv (p/q) \cdot q$. Unfortunately, however, there exists conflicts we need to take care of, which makes everything more complicated.

In an ideal world, we would expect the residual function to have type $D\ a \to D\ a \to Maybe\ (D\ a)$, where the partiality comes from receiving two non-aligned patches.

But what if 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 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;

- 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.

Above we see two distinct conflict types. 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!*

4.1 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 type of our residual[3]. operation is:

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

We reitarate that the partiality comes from the fact the residual is not defined for non-aligned patches. We chose to make a partial function instead of receiving a proof of alignment purely for pratical purposes. Defining alignment for our patches is very complicated.

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)))
```

[3] 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

```
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 $<M>$ denotes the Kleisli composition of the $Maybe$ monad and $\downarrow\!-map-\!\downarrow$ takes care of the indexes.

Now, we can compute both p/q and q/p at the same time. It also backs 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).

4.2 Solving Conflicts

- *This is highly dependent on the structure.*
  - *some structures might allow permutations, refactorings, etc... whereas others might not.*
- *How do we go generic? Free-monads to the rescue!*

5. Summary and Remarks

5.1 Sharing of Recursive Subterms

- If we want to be able to share recursive subexpressions we need a mutually recursive approach.
- Or, this will be handled during conflict solving. See refactoring.

5.2 Remarks on Type Safety

- only the interface to the user can be type-safe, otherwise we don't have our free-monad multiplication.

6. A Haskell Prototype

- *throw* hs-diff *in github before the deadline!*

7. 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...

8. 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!*

9. Conclusion

- This is what we take out of it.