Diffing Mutually Recursive Types

A code tour

Victor Cacciari Miraldo
University of Utrecht

December 16, 2016

1 Our Universe

The universe we are using is a *Sums-of-Products* over type variables and constant types.

```
data Atom (n : ℕ) : Set where
  I : Fin n      → Atom n
  K : Fin ks#    → Atom n
```

Constructor I refers to the n-th type variable whereas K refers to a constant type. Value $ks\#$ is passed as a module parameter. We denote products by π and sums by σ, but they are just lists.

```
π : ℕ → Set
π = List ∘ Atom

σπ : ℕ → Set
σπ = List ∘ π
```

Interpreting these codes is very simple. Here, Parms is a valuation for the type variables.

```
Parms : ℕ → Set₁
Parms n = Fin n → Set

⟦_⟧ₐ : {n : ℕ} → Atom n → Parms n → Set
⟦ I x ⟧ₐ      A = A x
⟦ K x ⟧ₐ      A = lookup x ks

⟦_⟧ₚ : {n : ℕ} → π n → Parms n → Set
⟦ [] ⟧ₚ       A = Unit
⟦ a :: as ⟧ₚ  A = ⟦ a ⟧ₐ A × ⟦ as ⟧ₚ A

⟦_⟧ : {n : ℕ} → σπ n → Parms n → Set
⟦ [] ⟧        A = ⊥
⟦ p :: ps ⟧   A = ⟦ p ⟧ₚ A ⊎ ⟦ ps ⟧ A
```

Note that here, Parms n really is isomorphic to n types that serve as the parameters to the functor $[\![F]\!]$. When we introduce a fixpoint combinator, these parameters are used to to tie the recursion knot, just like a simple fixpoint: $\mu\ F \equiv F\ (\mu F)$. In fact, a mutually recursive family can be easily encoded in this setting. All we need is n types that refer to n type-variables each!

```
Fam : ℕ → Set
Fam n = Vec (σπ n) n

data Fix {n : ℕ}(F : Fam n) : Fin n → Set where
  ⟨_⟩ : ∀{k} → ⟦ lookup k F ⟧ (Fix F) → Fix F k
```

This universe is enough to model Context-Free grammars, and hence, provides the basic bare bones for diffing elements of an arbitrary programming language. In the

future, it could be interesting to see what kind of diffing functionality indexed functors could provide, as these could have scoping rules and other advanced features built into them.

1.1 SoP peculiarities

One slightly cumbersome problem we have to circumvent is that the codes for type variables and constant types are modeled by Atoms; whereas the codes for types are modelled by $\sigma\pi$. This requires more discipline to organize our code, since we have to separate, on the type level, functions that handle one or the other. Nevertheless, we may wish to see Atoms as a trivial *Sum-of-Product*.

```
α : {n : ℕ} → Atom n → σπ n
α a = (a :: []) :: []
```

Instead of having binary injections into coproducts, like we would on a *regular-like* universe, we have n-ary injections, or, *constructors*. We encapsulate the idea of constructors of a $\sigma\pi$ into a type and write a *view* type that allows us to look at an inhabitant of a sum of products as a *constructor* and *data*.

First, we define constructors:

```
cons# : {n : ℕ} → σπ n → ℕ
cons# = length

Constr : {n : ℕ}(ty : σπ n) → Set
Constr ty = Fin (cons# ty)
```

Now, a constructor of type C expects some arguments to be able to make an element of type C. This is a product, we call it the typeOf the constructor.

```
typeOf : {n : ℕ}(ty : σπ n) → Constr ty → π n
typeOf [] ()
typeOf (x :: ty) fz  = x
typeOf (x :: ty) (fs c) = typeOf ty c
```

Injecting is fairly simple.

```
inject : {n : ℕ}{A : Parms n}{ty : σπ n}
  → (i : Constr ty) → ⟦ typeOf ty i ⟧p A
  → ⟦ ty ⟧ A
inject {ty = []} () cs
inject {ty = x :: ty} fz cs  = i1 cs
inject {ty = x :: ty} (fs i) cs = i2 (inject i cs)
```

We finish off with a *view* of $[\![ty]\!]_A$ as a constructor and some data. This greatly simplify the algorithms later on.

```
data SOP {n : ℕ}{A : Parms n}{ty : σπ n} : ⟦ ty ⟧ A → Set where
  strip : (i : Constr ty)(ls : ⟦ typeOf ty i ⟧p A)
    → SOP (inject i ls)
```

1.2 Agda Details

Here we clarify some Agda specific details that are agnostic to the big picture. This can be safely skipped on a first iteration.

As we mentioned above, our codes represent functors on n variables. Obviously, to program with them, we need to apply these to something. The denotation receives a function Fin $n \to$ Set, denoted Parms n, which can be seen as a valuation for each type variable.

In the following sections, we will be dealing with values of $[\![ty]\!]_A$ for some class of valuations A, though. We need to have decidable equality for $A\ k$ and some mapping from $A\ k$ to $\mathbb{N}$ for all k. We call such valuations a *well-behaved parameter*:

```
record WBParms {n : ℕ}(A : Parms n) : Set where
  constructor wb-parms
  field
    parm-size : ∀{k} → A k → ℕ
    parm-cmp  : ∀{k}(x y : A k) → Dec (x ≡ y)
```

TODO

The field *parm-size* is not really needed anymore! Remove it!

The following sections discuss functionality that does not depend on *parameters to codes*. Hence, we will be passing them as Agda module parameters. We also set up a number of synonyms to already fix the aforementioned parameter. The first diffing technique we discuss is the trivial diff. It's module is declared as follows:

```
module RegDiff.Diff.Trivial.Base
  {ks# : ℕ}(ks : Vec Set ks#)(keqs : Vecl Eq ks)
  {parms# : ℕ}(A : Parms parms#)(WBA : WBParms A)
  where
```

We stick to this nomenclature throughout the code. The first line handles constant types: $ks\#$ is how many constant types we have, ks is the vector of such types and $keqs$ is an indexed vector with a proof of decidable equality over such types. The second line handles type parameters: $parms\#$ is how many type-variables our codes will have, A is the valuation we are using and WBA is a proof that A is *well behaved*.

TODO

Now parameters are setoids, we can drop out the WBA record.

Below are the synonyms we use for the rest of the code:

```
  U : Set
  U = σπ parms#

  Atom : Set
  Atom = Atom' parms#

  Π : Set
  Π = π parms#

  sized : {p : Fin parms#} → A p → ℕ
  sized = parm-size WBA

  _≟-A_ : {p : Fin parms#}(x y : A p) → Dec (x ≡ y)
  _≟-A_ = parm-cmp WBA
```

```
⟦_⟧a : Atom → Set
⟦ a ⟧a = interpa a A

⟦_⟧p : Π → Set
⟦ p ⟧p = interpp p A

⟦_⟧ : U → Set
⟦ u ⟧ = interps u A

UUSet : Set1
UUSet = U → U → Set

AASet : Set1
AASet = Atom → Atom → Set

ΠΠSet : Set1
ΠΠSet = Π → Π → Set

contr : ∀{a b}{A : Set a}{B : Set b}
   → (A → A → B) → A → B
contr p x = p x x

UU→AA : UUSet → AASet
UU→AA P a a' = P (α a) (α a')

→α : {a : Atom} → ⟦ a ⟧a → ⟦ α a ⟧
→α k = i1 (k , unit)
```

2 Computing and Representing Patches

Intuitively, a *Patch* is some description of a transformation. Setting the stage, let A and B be types, $x : A$ and $y : B$ elements of such types. A *patch* between x and y must specify a few parts:

i) An $apply_p : A \to Maybe\ B$ function,

ii) such that $apply_p\ x \equiv \mathsf{just}\ y$.

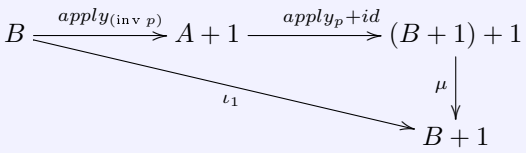
Well, $apply_p$ can be seen as a functional relation (R is functional iff $img\ R \subseteq id$ [so... it's a partial function! :) I'm probably guilty of suggesting relations instead of partial functions but, if we never make use of full-blown relations, then it's wiser to call a cat a cat. Besides, if we go to partial function, we could try to see how much of the Kleisli category we are making use of.)] [I agree... relations might be an overkill. I don't understand what you mean by "how much of the kleisli category we use". If I'm not mistaken, we use all of it! The products we get for free (Maybe monad is commutative), and the kleisli construction preserves coproducts]. [I think that this answers my question.]) from A to B. We call this the "application" relation of the patch, and we will denote it by $p^\flat \subseteq A \times B$.

Needs discussion:

There is still a lot that could be said about this. I feel like $p^\flat$ should also be invertible in the sense that:

i) Let (inv p) denote the inverse patch of p, which is a patch from B to A.

ii) Then, $p^\flat \cdot (\text{inv } p)^\flat \subseteq id$ and $(\text{inv } p)^\flat \cdot p^\flat \subseteq id$, Assuming $(\text{inv } p)^\flat$ is also functional, we can use the maybe monad to represent these relations in **Set**. Writing the first equation on a diagram in **Set**, using the *apply* functions:



iii) This is hard to play ball with. We want to say, in a way, that x $(p^\flat)$ y iff y $((\text{inv } p)^\flat)$ x. That is, (inv p) is the actual inverse of p. Using relations, one could then say that $(\text{inv } p)^\flat$ is the converse of $(p^\flat)$. That is: $(\text{inv } p)^\flat \equiv (p^\flat)^\circ$. But, if $(\text{inv } p)^\flat$ is functional, so is $(p^\flat)^\circ$. This is the same as saying that $p^\flat$ is entire! If $p^\flat$ is functional and entire, it is a function (and hence, total!). And that is not true.

[This is something we noticed in our ICFP'16 paper: if you ask for pointwise invertibility of partial functions, you end up asking for total functions. In fact, you want to define an order $\mathsf{nothing} \leqslant \_$ and $\mathsf{just}\ x \leqslant \mathsf{just}\ y$ iif $x = y$ on the monadic types. Then, the suitable notion of "equivalence" is a Galois connection, probably antitone in our case : $p >> p^{-1} \leqslant id$ and $p^{-1} >> p \leqslant id$.]
[I like the idea! I do not see the Galois connection arising, however. Which sets are we galois-connecting with which functors? This looks, however, exactly what I was looking for!]
[Let $f : A \rightharpoonup B$ and $g : B \rightharpoonup A$. We have $f >> g : B \rightharpoonup B$ and $g >> f : A \rightharpoonup A$ by Kleisli composition. We say that f and g form an antitone Galois connection if $\forall a : \mathsf{Maybe}\ A, (g >> f)(a) \leqslant a$ and $\forall b : \mathsf{Maybe}\ B, (f >> g)(b) \leqslant b$.]
[And, indeed, when you define the interpretation of patches as relations throughout the report, your intuition seems really to be about such pairs of partial maps. Rather that give tricky relations, it may be simpler to just go with pairs of somewhat invertible partial functions. The story about the ordering of patches carries over to this case: this amounts to lifting $\leqslant$ above pointwise to functions.]
[I'm not sure I see this lifting happening. Let leq_f be such lifted relation. We would then say that a patch p is better than a patch q, that is, $q \leqslant p$ iff $\forall x.q\ x \leqslant p\ x$. Which basically translates to p is defined, at least, for every x that q is also defined. Which implies a bigger domain. It makes sense! yes!] [yes, that's what I meant: sorry, I should have been more explicit.]
[Aside from this technicality, I find the whole framework aesthetically unpleasing: we are specifying the function "apply" over a patch computed over two elements and their relative coherence. It feels like their is some structure we are not exploiting at these different levels.]

Needs discussion:

[Inspired by Tabareau's "Aspect-oriented programming: a language for 2-categories", I'm toying with (but not proposing for adoption, this is still very sketchy!) the following idea: we define a 2-category where 0-cells are types, 1-cells are antitone Galois connections (pairs of partial functions) (ie. diff and inverse diff) between types and 2-cells are residuals. There is a terminal object "unit": it is a 0-cell 1 such that for all type A, there exists a (trivial) 1-cell $\mathsf{skip} : A \to 1$ and a unique 2-cell $1_{\mathsf{skip}} : \mathsf{skip} \Rightarrow \mathsf{skip}$. The specification of "patch" becomes: for every 1-cell $x : 1 \to A$ and $y : 1 \to B$ there exists a 1-cell $p : A \to B$ and a 2-cell $\mathsf{apply} : x \Rightarrow y$. Take this with a pinch of salt: it is more "wishful thinking" than "sound categorical reasoning".]

[I like the sketch, let me know where this goes! By the way, not sure we need to carry around the pair of partial functions. The inverse diff is uniquely determined by the diff!]

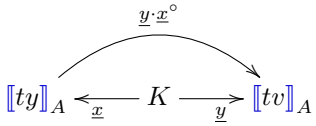
[re "inverse diff uniquely determined by the diff": I don't think that this is true. You could for example provide an inverse diff that fails all the time: it would satisfy our spec above. However, given a diff code, you can always compute *one* specific inverse. (Which, btw, should have some sort of universal property that we ought to characterize!)]

Now, let us discuss some code and build some intuition for what is what in the above schema. We will present different parts of the code, how do they relate to this relational view and give examples here and there!

2.1 Trivial Diff

The simplest possible way to describe a transformation is to say what is the source and what is the destination of such transformation. This can be accomplished by the Diagonal functor, Δ, just fine.

Now, take an element $(x\ ,\ y)\ :\ \Delta\ ty\ tv$. The "application" relation it defines is trivial: $\{(x, y)\}$, or, in point-free style:



Where, for any $A, B \in Set$ and $x : A$, $\underline{x} \subseteq A \times B$ represents the *everywhere* x relation, defined by

$$\underline{x} = \{(x, b) \mid b \in B\}$$

This is a horrible patch however: We can't calculate with it because we don't know *anything* about *how* x changed into y. Note, however, that $(x\ ,\ y)^\flat \equiv \underline{y} \cdot \underline{x}^\circ$ is trivially functional.

Needs discussion:

In the code, we actually define the "application" relation of Δ as:

$$(x\ ,\ x)^\flat = id$$
$$(x\ ,\ y)^\flat = \underline{y} \cdot \underline{x}^\circ$$

This suggests that copies might be better off being handled by the trivial diff. We will return to this discussion in section 3

2.1.1 Trivial Diff, in Agda

We will be using $\Delta\ ty\ tv$ for the three levels of our universe: atoms, products and sums. We distinguish between the different Δ's with subscripts $_a$, $_p$ and $_s$ respectively. They only differ in type. The treatment they receive in the code is exactly the same! Below is how they are defined:

```
delta : ∀{a}{A : Set a}(P : A → Set)
  → A → A → Set
delta P a1 a2 = P a1 × P a2
```

Hence, we define $\Delta_x = \mathsf{delta}\ [\![\ \cdot\ ]\!]_x$, for $x \in \{a, p, s\}$.

2.2 Spines

We can make the trivial diff better by identifying whether or not x and y agree on something! In fact, we will aggresively look for copying oportunities. We start by checking if x and y are, in fact, equal. If they are, we say that the patch that transforms x into y is *copy*. If they are not equal, they might have the same *constructor*. If they do, we say that the constructor is copied and we put the data side by side (zip). Otherwise, there is nothing we can do on this phase and we just return $\Delta\ x\ y$.

Note that the *spine* forces x and y to be of the same type! In practice, we are only interested in diffing elements of the same language. It does not make sense to diff a C source file against a Haskell source file.

[Even in a single language, it would not make sense to *even try* to match terms of different types. This would only contribute to a combinatorial explosion while yielding potentially ill-typed transformations. This is one of those places where we could argue for improved efficiency of the type-directed approach (if it actually pays out in practice)]

[Very intesting: by going to a sum-of-product presentation, you've stratified the positive (sums) and negative (product) types. Which, indeed, immensely simplifies the computation of the spine: we may have choices in sums and must go right across products (through ListI).]

Nevertheless, we define an S structure to capture this longest common prefix of x and y; which, for the SoP universe is very easy to state.

```
data S (P : UUSet) : U → Set where
  SX  : {ty : U} → P ty ty → S P ty
  Scp : {ty : U} → S P ty
  Scns : {ty : U}(i : Constr ty)
       → Listl (contr P ∘ α) (typeOf ty i)
       → S P ty
```

Remember that $\mathsf{contr}\ P\ x\ =\ P\ x\ x$ and $\alpha\ :\ \mathsf{Atom}\ n\ \to\ \sigma\pi\ n$; Here, $\mathsf{Listl}\ P\ l$ is an indexed list where the elements have type $P\ l_i$, for every $l_i \in l$. We will treat this type like an ordinary list for the remainder of this document.

Note that S makes a functor (actually, a free monad!) on P, and hence, we can map over it:

```
S-map : {ty : U}
        {P Q : UUSet}(X : ∀{k v} → P k v → Q k v)
      → S P ty → S Q ty
S-map f (SX x)      = SX (f x)
S-map f Scp         = Scp
S-map f (Scns i xs) = Scns i (map_i f xs)
```

Computing a spine is easy, first we check whether or not x and y are equal. If they are, we are done. If not, we look at x and y as true sums of products and check if their constructors are equal, if they are, we zip the data together. If they are not, we zip x and y together and give up.

[Note that zipping here is accelerating the diffing computation: since the data is well-typed, they **have to** have the same arity and this is **not** an alignment problem.

In an untyped setting, we would have to be paranoid and find a matching sequence.]
[Very true! In fact, this is how we force looking for the *largest common prefix* in the fixpoint case].

```
spine-cns : {ty : U}(x y : ⟦ ty ⟧) → S Δs ty
spine-cns x y   with sop x | sop y
spine-cns _ _ | strip cx dx | strip cy dy
   with cx ≟-Fin cy
...| no  _       = SX (inject cx dx , inject cy dy)
spine-cns _ _ | strip _ dx | strip cy dy
   | yes refl    = Scns cy (zipp dx dy)

spine : {ty : U}(x y : ⟦ ty ⟧) → S Δs ty
spine {ty} x y
   with dec-eq _≟-A_ ty x y
...| yes _       = Scp
...| no  _       = spine-cns x y

zipp : {ty : Π}
   → ⟦ ty ⟧p → ⟦ ty ⟧p → Listl (λ k → Δs (α k) (α k)) ty
zipp {[]}  _  _
   = []
zipp {_ :: ty} (x , xs) (y , ys)
   = (i1 (x , unit) , i1 (y , unit)) :: zipp xs ys
```

The "application" relations specified by a spine $s = \mathsf{spine}\ x\ y$, denoted $s^\flat$ are defined by:

$$\mathsf{Scp}^\flat = \quad A \xleftarrow{\ id\ } A$$

$$(\mathsf{SX}\ p)^\flat = \quad A \xleftarrow{\ p^\flat\ } A$$

$$(\mathsf{Scns}\ i\ [s1\ ,\ \cdots\ ,\ sN])^\flat = \quad \Pi_k\Pi_j A_{kj} \xleftarrow{\ \mathsf{inj}_i \cdot (s_1{}^\flat \times \cdots \times s_n{}^\flat) \cdot \mathsf{inj}_i{}^\circ\ } \Pi_k\Pi_j A_{kj}$$

where inj_i is the injection, with constructor i, into $\Pi_k T_k$. It corresponds to the relational lifting of function injection.

Note that, in the ($\mathsf{SX}\ p$) case, we simply ask for the "application" relation of p. The algorithm produces a $\mathsf{S}\ \Delta_s$, so we have pairs on the leaves of the spine. In fact, either we have only one leave or we have *arity* C_i leaves, where C_i is the common constructor of x and y in $\mathsf{spine}\ x\ y$.

For a running example, let's consider a datatype defined by:

```
2-3-TREE-F : σπ 1
2-3-TREE-F  = []
            ⊕ (K kℕ) ⊗ I ⊗ I ⊗ []
            ⊕ (K kℕ) ⊗ I ⊗ I ⊗ I ⊗ []
            ⊕ []
```

We ommit the fz for the I parts, as we only have one type variable. We also use _⊕_ and _⊗_ as aliases for _::_ with different precedences. As expected, there are three constructors:

```
2-node' 3-node' nil' : Constr 2-3-TREE-F
nil'    = fz
2-node' = fs fz
3-node' = fs (fs fz)
```

We can then consider a few spines over $[\![\text{2-3-TREE-F}]\!]_{\text{Unit}}$ to illustrate the algorithm:

```
spine nil' (3-node' 10 unit unit unit) = SX (nil' , 3-node' 10 unit unit unit)
spine (2-node' 10 unit unit) (2-node' 15 unit unit) = Scns 2-node' [ (10 , 15) , (unit , unit) , (unit , unit) ]
spine nil' nil' = Scp
```

In the case where the spine is Scp or Scns i there is nothing left to be done and we have the best possible diff. Note that on the Scns i case we do *not* allow for rearanging of the parameters of the constructor i.

In the case where the spine is SX, we can do a better job! We can record which constructor changed into which and try to reconcile the data from both the best we can. Going one step at a time, let's first change one constructor into the other.

It is important to note that if the output of spine is a SX, then the constructors are *different*.

2.3 Constructor Changes

Let's take an example where the spine can not copy anything:

```
s = spine (2-node' 10 unit unit) (3-node' 10 unit unit unit)
  = SX (2-node' 10 unit unit , 3-node' 10 unit unit unit)
```

Here, we wish to say that we changed a 2-node' into a 3-node'. But we are then left with a problem about what to do with the data inside the 2-node' and 3-node'; this is where the notion of alignment will be in the picture. For now, we abstract it away by the means of a parameter, just like we did with the S. This time, however, we need something that receives products as inputs.

```
data C (P : ΠΠSet) : U → U → Set where
  CX  : {ty tv : U}
      → (i : Constr ty)(j : Constr tv)
      → P (typeOf ty i) (typeOf tv j)
      → C P ty tv
```

Note that C also makes up a functor, and hence can be mapped over:

```
C-map  :  {ty tv : U}
          {P Q : ΠΠSet}(X : ∀{k v} → P k v → Q k v)
       → C P ty tv → C Q ty tv
C-map f (CX i j x) = CX i j (f x)
```

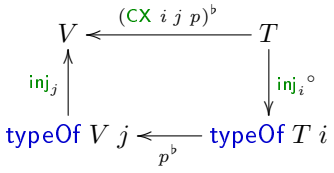
Computing an inhabitant of C is trivial:

```
change : {ty tv : U} → [[ ty ]] → [[ tv ]] → C Δp ty tv
change x y with sop x | sop y
change _ _ | strip cx dx | strip cy dy = CX cx cy (dx , dy)
```

Now that we can compute change of constructors, we can refine our s above. We can compute S-map change s and we will have:

```
c = S-map change s
  = SX (CX 2-node' 3-node' ((10 , unit , unit) , (10 , unit , unit , unit)))
```

The "application" relation induced by C is trivial. We just need to pattern match, change the data of the constructor in whatever way we need, then inject into another type.



Note that up until now, everything was deterministic! This is something we are bound to lose when talking about alignment.

2.4 Aligning Everything

On the literature for version control system, the *alignment* problem is the problem of mapping two strings l_1 and l_2 in $\mathcal{L}$ into $\mathcal{L} \cup \{-\}$, for $\{-\} \notin \mathcal{L}$ such that the resulting strings l_1' and l_2' are of the same length such that for all i, it must not be happen that $l_1'[i] = - = l_2'[i]$. For example, Take strings $l_1 = "CGTCG"$ and $l_2 = "GATAGT"$, then, the following is an (optimal) alignment:

C	G	-	T	C	G	-
-	G	A	T	A	G	T

Let $\mathcal{DNA} = \{A, T, C, G\}$. Finding the table above is the same as finding a partial map:

$$f : \mathcal{DNA}^5 \to \mathcal{DNA}^6$$

such that $f\ (C, G, T, C, G) = (G, A, T, A, G, T)$. There are many ways of defining such a map. We would like, however, that our definition have a maximal domain, that is, we impose the least possible amount of restrictions. In this case, we can actually define f with some pattern matching as:

$$\begin{array}{lll} f & (C, x, y, C, z) & = (x, A, y, A, z, T) \\ f & \_ & = \text{undefined} \end{array}$$

And it is easy to verify that, in fact, $f\ (C, G, T, C, G) = (G, A, T, A, G, T)$. Moreover, this is the *maximal* such f that still (provably) assigns the correct destination to the correct source.

On our running example, the leaf of c has type Δ_p (*typeOf* 2-node‘) (*typeOf* 3-node‘), and it's value is ((*10* , unit , unit) , (*10* , unit , unit , unit)). Note that we are now dealing with products of different arity. This step will let us say how to *align* one with the other!

On our example, as long as we align the 10 with the 10, the rest does not matter. One optimal alignment could be:

10	–	unit	unit
10	unit	unit	unit

2.4.1 Back to Agda

We will look at alignments from the "finding a map between products" perspective. Here is where our design space starts to grow, and so, we should start making some distinctios:

- We want to allow sharing. This means that the there can be more than one variable in the defining pattern of our f.
- We do *not* allow permutations, as the search space would be too big. This means that the variables appear in the right-hand side of f in the same order as they appear in the left-hand-side.
- We do *not* allow contractions nor weakenings. That is, every variable on the left-hand-side of f must appear *exactly* once on the right-hand-side.

[Fascinating! We are back to the world of good ol' diff: matching lists of objects. Performance-wise, this means that we should have the same asymptotic complexity and that we may have a chance to be more efficient in practice.]

[Absolutely! I'm currently looking at some Dynamorphism techniques to write this in Agda. Worst case scenario, in Haskell, we stick to memoization].

[Talking about performance, the theory says that, once the changes have been computed, we could solve all the resulting alignment problems in parallel. How easy could you implement this in your Haskell proto? Any noticeable speed-up/slow-down?]

[I'm not aware of such theory. From what I know, computing an optimal alignment and an optimal diff is the same thing (for untyped trees). I don't understand what you mean by computing the alignment AFTER the changes have been computed. In our algorithm, at least, this happens at the same time.].

[re "theory": the positive type/negative type distinction I keep referring to and which drives my intuition is used to structure proof search in linear logic. There, the product would be called an *asynchronous* connective while the sum would be called a *synchronous* connective. Quoting "Focusing and Polarization in Intuitionistic Logic", "the search for a focused proof can capitalize on this classification by applying [..] all invertible rules [related to an asynchronous connective] in any order (without the need for backtracking) and by applying a chain of non-invertible rules [related to a synchronous connective] that focus on a given formula and its positive subformulas.". So my gut tells me that your diff computation is structured in two (repeating) phases: one that generates the spine & change, yielding several **independant** alignment problems which could be solved concurrently. Is that clearer?]

The following datatype describe such maps:

```
data Al (P : AASet) : Π → Π → Set where
  A0  :                                          Al P [] []
  Ap1 : ∀{a ty tv}     → ⟦ a ⟧a → Al P ty tv →  Al P (a :: ty) tv
  Ap1° : ∀{a ty tv}    → ⟦ a ⟧a → Al P ty tv →  Al P ty (a :: tv)
  AX  : ∀{a a' ty tv}  → P a a' → Al P ty tv →  Al P (a :: ty) (a' :: tv)
```

Note that the indexes of Al, although represented as lists are, in fact, products. Well, turns out that lists and products are not so different after all. Let us represent the f we devised on the $\mathcal{DNA}$ example using Al. Recall $f\ (C, x, y, C, z) = (x, A, y, A, z, T)$.

```
f ≡ Ap1 C (AX Scp (Ap1° A (AX Scp (AX (C , A) (AX Scp (Ap1° T A0))))))
```

If we rename Ap1 to *del*; Ap1° to *ins* and AX to *mod* we see some familiar structure arising! Aligning products is the same as computing the diff between heterogeneous lists! In fact, the align function is defined as:

```
align* : {ty tv : Π} → ⟦ ty ⟧p → ⟦ tv ⟧p → List (Al Δa ty tv)
align* {[]}  {[]}      m n = return A0
align* {[]}  {v :: tv} m (n , nn)
  = Ap1° n <$> align* m nn
align* {y :: ty} {[]}  (m , mm) n
  = Ap1 m <$> align* mm n
align* {y :: ty} {v :: tv} (m , mm) (n , nn)
  =  AX (m , n)  <$> align* mm nn
  ++ Ap1  m      <$> filter (not ∘ is-ap1°)  (align* mm (n , nn))
  ++ Ap1° n      <$> filter (not ∘ is-ap1)   (align* (m , mm) nn)
  where
    is-ap1 : {ty tv : Π} → Al Δa ty tv → Bool
    is-ap1 (Ap1 _ _) = true
    is-ap1 _  = false

    is-ap1° : {ty tv : Π} → Al Δa ty tv → Bool
    is-ap1° (Ap1° _ _) = true
    is-ap1° _  = false
```

We are now doing things in the *List* monad. This is needed because there are many possible alignments between two products. For the moment, we refrain from choosing and compute all of them.

On another note, some of these alignments are simply dumb! We do not want to have both Ap1 x (Ap1° y a) and AX (x , y) a. They are the same alignment. The filters are in charge of pruning out those branches from the search space. [this strikes me as a rather wasteful. I'm under the impression that you could probably enforce the absence of 'ap1' or 'ap1o' at the type level and dispense from generating these cases in the first place.]

[We could use a table type, as Lempsink did in "Type-safe diff for families of types". This is very cumbersome though. My plan is to use dynamorphisms to structure the recursion like it should be. The filters are there just to make the agda prototype compute faster.]

Sticking with our example, we can align the leaves of our c by computing the following expression, where C-mapM is simply the monadic variant of C-map.

$$
\begin{aligned}
a &= \mathsf{C\text{-}mapM}\ \mathsf{align}^*\ c \\
&= \mathsf{SX}\ (\mathsf{CX}\ \mathsf{2\text{-}node}`\ \mathsf{3\text{-}node}`\ (\mathsf{AX}\ (10\ ,\ 10)\ (\mathsf{AX}\ (\mathsf{unit}\ ,\ \mathsf{unit})\ \cdots))) \\
&\ \ ::\mathsf{SX}\ (\mathsf{CX}\ \mathsf{2\text{-}node}`\ \mathsf{3\text{-}node}`\ (\mathsf{Ap1}\ 10\ (\mathsf{AX}\ (\mathsf{unit}\ ,\ 10)\ \cdots))) \\
&\ \ ::\mathsf{SX}\ (\mathsf{CX}\ \mathsf{2\text{-}node}`\ \mathsf{3\text{-}node}`\ (\mathsf{Ap1}\ 10\ (\mathsf{Ap1}\ \mathsf{unit}\ \cdots))) \\
&\ \ ::\mathsf{SX}\ (\mathsf{CX}\ \mathsf{2\text{-}node}`\ \mathsf{3\text{-}node}`\ (\mathsf{Ap1°}\ 10\ (\mathsf{AX}\ (10\ ,\ \mathsf{unit})\ \cdots))) \\
&\ \ \cdots
\end{aligned}
$$

Now we have a problem. Which of the patches above should we chose to be *the* patch? Recall that we mentioned that we wanted to find the alignment with *maximum domain*. Something interesting happens if we look at patches from their "application" relation, but first, we define the "application" relations of Al:

$$
\begin{aligned}
(\mathsf{AX}\ a_1\ a_2)^\flat &= B \times \Pi D \xleftarrow{{a_1}^\flat \times {a_2}^\flat} A \times \Pi C \\
(\mathsf{Ap1}\ x\ a)^\flat &= \Pi B \xleftarrow{\langle \underline{x}, {a^\flat}^\circ \rangle^\circ} X \times \Pi A \\
(\mathsf{Ap1°}\ x\ a)^\flat &= X \times \Pi B \xleftarrow{\langle \underline{x}, a^\flat \rangle} \Pi A \\
\mathsf{A0}^\flat &= \mathbb{1} \xleftarrow{\top} \mathbb{1}
\end{aligned}
$$

3 Patches as Relations

In order to better illustrate this concept, we need a simpler example first. Let's consider the following type with no type variables:

```
Type1 = K kℕ ⊗ []
      ⊕ K kℕ ⊗ K kℕ ⊗ []
      ⊕ []
```

It clearly has two constructors:

```
C₁ C₂ : Constr Type1
C₁    = fz
C₂    = fs fz
```

Now, let's take two inhabitants of Type1.

```
x y : ⟦ Type1 ⟧
x   = inject C₂ (4 , 10 , unit)
y   = inject C₁ (10 , unit)
```

There are two possible options for diff x y:

```
ds : Patch* Type1                                                    .
ds = SX (CX C2 C1 (AX  (4 , 10) (Ap1 10       A0))) - P1
   :: SX (CX C2 C1 (Ap1 4       (AX  (10 , 10) A0))) - P2
   :: []
```

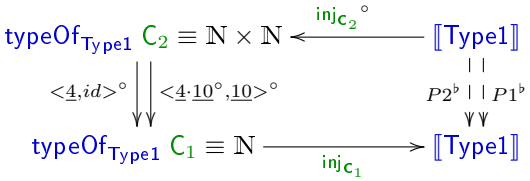
Consider the semantics for Δ as described in the discussion box at Section 2.1, that is,

$$(x\ ,\ y)^\flat = \begin{cases} id & \text{if } x \equiv y \\ \underline{y} \cdot \underline{x}^\circ & \text{otherwise} \end{cases}$$

Then it becomes clear that we want to select patch (P2) instead of (P1). In fact, there is a deeper underlying reason for that! Looking at the two patches as relations (after some simplifications), we have:

$$P1^\flat = \mathsf{inj}_{C_1} \cdot \langle (\underline{4} \cdot \underline{10}^\circ), \underline{10} \rangle^\circ \cdot \mathsf{inj}_{C_2}{}^\circ$$
$$P2^\flat = \mathsf{inj}_{C_1} \cdot \langle \underline{4}, id \rangle^\circ \cdot \mathsf{inj}_{C_2}{}^\circ$$

Drawing them in a diagram we have:



Here, we have something curious going on... We have that $P1^\flat \subseteq P2^\flat$. To see this is not very hard. First, composition and converses are monotonous with respect to $\subseteq$. We are left to check that:

$$\begin{array}{ll} & < \underline{4} \cdot \underline{10}^\circ, \underline{10} > \subseteq < \underline{4}, id > \\ \equiv & \{ \text{ split universal } \} \\ & \pi_1 \cdot < \underline{4} \cdot \underline{10}^\circ, \underline{10} > \subseteq \underline{4} \ \wedge \ \pi_2 \cdot < \underline{4} \cdot \underline{10}^\circ, \underline{10} > \subseteq id \end{array}$$

The first proof obligation is easy to calculate with:

$$\begin{array}{ll} & \pi_1 \cdot < \underline{4} \cdot \underline{10}^\circ, \underline{10} > \subseteq \underline{4} \\ \Leftarrow & \{ \ \pi_1\text{-cancel} \ ; \ \subseteq\text{-trans} \ \} \\ & \underline{4} \cdot \underline{10}^\circ \subseteq \underline{4} \\ \Leftarrow & \{ \ \text{Leibniz} \ \} \\ & \underline{4} \cdot \underline{10}^\circ \cdot \underline{10} \subseteq \underline{4} \cdot \underline{10} \\ \equiv & \{ \ \underline{a}^\circ \cdot \underline{a} \equiv \top \ \} \\ & \underline{4} \cdot \top \subseteq \underline{4} \cdot \underline{10} \\ \equiv & \{ \ \underline{a} \cdot \underline{b} \equiv \underline{a} \ \} \\ & \underline{4} \cdot \top \subseteq \underline{4} \\ \equiv & \{ \ \underline{a} \cdot \top \equiv \underline{a} \ \} \\ & \underline{4} \subseteq \underline{4} \\ \equiv & \{ \ \subseteq\text{-refl} \ \} \\ & True \end{array}$$

The second is easier to prove once we add variables!

$$
\begin{array}{ll}
& \pi_2 \cdot < \underline{4} \cdot \underline{10}^\circ, \underline{10} > \subseteq \ \ id \\
\equiv & \{ \ \text{Add variables} \ \} \\
& \forall x, y \ . \ x \ (\pi_2 \cdot < \underline{4} \cdot \underline{10}^\circ, \underline{10} >) \ y \Rightarrow x = y \\
\equiv & \{ \ \text{PF expand composition} \ \} \\
& \forall x, y \ . \exists z \ . \ x \ (\pi_2) \ z \wedge z \ < \underline{4} \cdot \underline{10}^\circ, \underline{10} > \ y \Rightarrow x = y \\
\equiv & \{ \ \text{Types force } z = (z_1, z_2) \ \} \\
& \forall x, y \ . \exists z_1, z_2 \ . \ x \ (\pi_2) \ (z_1, z_2) \wedge (z_1, z_2) \ < \underline{4} \cdot \underline{10}^\circ, \underline{10} > \ y \Rightarrow x = y \\
\equiv & \{ \ \pi_2 \ \text{def} \ \} \\
& \forall x, y \ . \exists z_1, z_2 \ . \ x = z_2 \wedge (z_1, z_2) \ < \underline{4} \cdot \underline{10}^\circ, \underline{10} > \ y \Rightarrow x = y \\
\equiv & \{ \ \text{split def} \ \} \\
& \forall x, y \ . \exists z_1, z_2 \ . \ x = z_2 \wedge z_1 \ (\underline{4} \cdot \underline{10}^\circ) \ y \wedge z_2 \ (\underline{10}) \ y \Rightarrow x = y \\
\equiv & \{ \ \text{points def} \ \} \\
& \forall x, y \ . \exists z_1, z_2 \ . \ x = z_2 \wedge z_1 = 4 \wedge y = 10 \wedge z_2 = 10 \Rightarrow x = y \\
\equiv & \{ \ \text{substitutions ; weakening} \ \} \\
& \forall x, y \ . \exists z_2 \ . \ x = 10 \wedge y = 10 \Rightarrow x = y \\
\equiv & \{ \ \text{trivial} \ \} \\
& True
\end{array}
$$

Nevertheless, it is clear which patch we should choose! We should always choose the patch that gives rise to the biggest relation, as this is applicable to much more elements.

Hence, our *cost* functions will count how many elements of the domain and range of the "application" relation of a patch are *fixed*. Note that the S and C parts of the algorithm are completely deterministic, hence they should *not* contribute to cost:

```
S-cost : {ty : U}{P : UUSet}(doP : {k v : U} → P k v → ℕ)
  → S P ty → ℕ
S-cost doP (SX x)  = doP x
S-cost doP Scp     = 0
S-cost doP (Scns i xs) = foldr_i (λ h r → doP h + r) 0 xs

C-cost  : {ty tv : U}{P : ΠΠSet}(doP : {k v : Π} → P k v → ℕ)
        → C P ty tv → ℕ
C-cost doP (CX i j x) = doP x
```

An Alignment might fix one element on the source, using Ap1 or one element on the destination, using Ap1°.

```
Al-cost : {ty tv : Π}{P : AASet}(doP : {k v : Atom} → P k v → ℕ)
  → Al P ty tv → ℕ
Al-cost doP A0        = 0
Al-cost doP (Ap1 x a) = 1 + Al-cost doP a
Al-cost doP (Ap1° x a) = 1 + Al-cost doP a
Al-cost doP (AX x a)  = doP x + Al-cost doP a
```

Last but not least, a Δ will either fix 2 elements: one in the source that becomes one in the destination; or none, when we just copy the source.

```
cost-delta-raw : ℕ
cost-delta-raw = 2

cost-delta : ∀{α}{A : Set α}{ty tv : A}(P : A → Set)
   (eqA : (x y : A) → Dec (x ≡ y))
   (eqP : (k : A)(x y : P k) → Dec (x ≡ y))
→ delta P ty tv → ℕ
cost-delta {ty = ty} {tv = tv} P eqA eqP (pa1 , pa2)
   with eqA ty tv
...| no _ = cost-delta-raw
cost-delta {ty = ty} P eqA eqP (pa1 , pa2)
   | yes refl with eqP ty pa1 pa2
...| no  _ = cost-delta-raw
...| yes _ = 0
```

According to these definitions, the cost of (P1) above is 3, where the cost of (P2) is 1.

3.1 Patches for Regular Types

Now that we have *spines*, *changes* and *alignments* figured out, we can define a patch as:

```
Patch : AASet → U → Set
Patch P = S (C (Al P))
```

Computing inhabitants of such type is done with:

```
diff1* : {ty : U}(x y : ⟦ ty ⟧) → Patch* ty
diff1* x y = S-mapM (C-mapM (uncurry align*) ∘ uncurry change) (spine x y)
```

where Patch* is defined as List (Patch Δ_a).

3.2 Conjectures About the cost function

Here we conjecture a few lemmas about the interplay of the cost function and the "application" relation. Let P, Q and R be patches.

i) If P has a lower cost than Q, then the domain and range of the "application" relation of P contains the "application" relation of Q.

$$\mathsf{cost}\ P < \mathsf{cost}\ Q \ \Rightarrow\ Q^\flat \subseteq P^\flat \cdot \top \cdot P^\flat$$

Needs discussion:

This is not as simple as

$$\mathsf{cost}\ P < \mathsf{cost}\ Q \ \Rightarrow\ Q^\flat \subseteq P^\flat$$

Take two *Deltas*, $px = (10\ ,\ 50)$ and $py = (30\ ,\ 30)$. Trivially, $\mathsf{cost}\ py = 0$ and $\mathsf{cost}\ px = 2$. Now, $px^\flat = \underline{50} \cdot \underline{10}^\circ$ and $py^\flat = id$. It is not true that $\underline{50} \cdot \underline{10}^\circ \subseteq id$!
If we state, however: Let P , Q in $diff* x\ y$; $\mathsf{cost}\ P < \mathsf{cost}\ Q \ \Rightarrow\ Q^\flat \subseteq P^\flat$ Seems more likely. As the above counter example would not work anymore. $diff*\ 10\ 50 = (10\ ,\ 50) :: []$.

ii) If P and Q have equal cost, it means that there is at least one place where P and Q are doing *the same thing*, hence there is a patch that copies this *same thing* and costs strictly less.

$$\mathsf{cost}\ P \equiv \mathsf{cost}\ Q \ \Rightarrow\ \exists\ R \cdot \mathsf{cost}\ R < \mathsf{cost}\ P$$

4 Mutually Recursive Types

Now that we have a clear picture of regular types, extending this to recursive types is not very difficult.

First, recall that a mutually recursive family is defined as n codes that each reference n type variables:

```
Fam : ℕ → Set
Fam n = Vec (σπ n) n

data Fix {n : ℕ}(F : Fam n) : Fin n → Set where
  ⟨_⟩ : ∀{k} → ⟦ lookup k F ⟧ (Fix F) → Fix F k
```

Another auxiliary definition we use here is the indexed coproduct, which let's us *extend* some indexed type.

```
_+ᵤ_ : {n : ℕ} → (σπ n → σπ n → Set) → (σπ n → σπ n → Set) → (σπ n → σπ n → Set)
(P +ᵤ Q) ty tv = (P ty tv) ⊎ (Q ty tv)
```

Now, we already have the ingredients for detecting and representing common constructors or full copies, with S, constructor changes, with C and alignments with Al. We just need to handle type variables to tie the knot. Before we proceed with the nasty definitions, we still need two last synonyms:

```
Famᵢ : Set
Famᵢ = Fin fam#

T : Famᵢ → σπ fam#
T k = lookup k fam
```

Here T k represents the k-th type of the family, and Fam_i acts a the the types in the family.

We start defining a patch for fixed points by allowing normal patches to perform changes on the first layer.

```
data Patchμ : U → U → Set where
  skel : {ty : U} → Patch (UU→AA Patchμ) ty
       → Patchμ ty ty
```

However, a patch for a fixed point might not only follow the precise order of operations (S, then C, then Al) that regular types enjoyed. For instance, imagine we are transforming the following lists:

$$[5, 8, 13, 21] \rightsquigarrow [8, 13, 21]$$

Let lists be seen (as usual) as the initial algebras of $L_A\ X = 1 + A \times X$; then, both lists are inhabitants of $\mu L_{\mathbb{N}}$, but, more precisely, the source is an inhabitant of $L_{\mathbb{N}}(L_{\mathbb{N}}(L_{\mathbb{N}}(L_{\mathbb{N}}\ \mathbb{1})))$ whereas the target is an inhabitant of $L_{\mathbb{N}}(L_{\mathbb{N}}(L_{\mathbb{N}}\ \mathbb{1}))$.

Here, we are already beginning with different types, so a spine (which is homogeneous) might not be the best start! In fact, the best start is to say that the first 5 is deleted, then the spine can kick in and say that everything else is copied!

Deleting and inserting can be seen as alignments between a type variable and the type we wish to delete or insert. For instance, deleting 5, which is actually $5 :: \_$, consists in deleting the $\_ :: \_$ constructor and aligning the $\mathbb{N} \times \mathsf{I}\ k$ with with a type variable $\mathsf{I}\ k$. Insertions are analogous.

```
ins : {ty : U}{k : Famᵢ}(i : Constr ty)
    → Al (UU→AA Patchμ) (I k :: []) (typeOf ty i)
    → Patchμ (T k) ty
del : {ty : U}{k : Famᵢ}(i : Constr ty)
    → Al (UU→AA Patchμ) (typeOf ty i) (I k :: [])
    → Patchμ ty (T k)
```

Note, however, that we use UU→AA Patchμ to populate the leaves of Patch and Al. That's because their parameter is of type Atom → Atom → *Set*, but Patchμ has type

$U \to U \to Set$. Nevertheless, when these leaves are just two type variables, we want to keep using Patchμ to record their differences. When these leaves are constant types, we give up and set their values with a delta.

```
fix : {k k' : Fam_i}
    → Patchμ (T k) (T k')
    → Patchμ (α (I k)) (α (I k'))
set : {ty tv : U}
    → Δ_s ty tv
    → Patchμ ty tv
```

Associating costs are trivial. We just piggy back on the previous cost definitions. I am still very puzzled for how this works.

```
{-# TERMINATING #-}
Patchμ-cost : {ty tv : U} → Patchμ ty tv → ℕ
Patchμ-cost (skel x)
  = Patch-cost Patchμ-cost x
Patchμ-cost (ins i x)
  = Al-cost Patchμ-cost x
Patchμ-cost (del i x)
  = Al-cost Patchμ-cost x
Patchμ-cost (fix p)
  = Patchμ-cost p
Patchμ-cost (set x)
  = cost-Δ_s x
```

Nevertheless, the heart of the algorithm is the same as any patching algorithm out there. Things can be modified, inserted or deleted:

```
{-# TERMINATING #-}
diffμ* : {k k' : Fam_i} → Fix fam k → Fix fam k' → List (Patchμ (T k) (T k'))
diffμ* {k} {k'} ⟨ x ⟩ ⟨ y ⟩
  =    diffμ*-mod {T k}          {T k'}  x    y
  ++ diffμ*-ins {lookup k' fam} {k}    ⟨ x ⟩  y
  ++ diffμ*-del {lookup k fam}  {k'}   x  ⟨ y ⟩
```

Modifying must happen at the same type only, otherwise we force an insertion or deletion. If we are, in fact, on the same type, we can get the patch for that layer and continue diffing atoms.

```
diffμ*-mod : {ty tv : U} → ⟦ ty ⟧ → ⟦ tv ⟧ → List (Patchμ ty tv)
diffμ*-mod {ty} {tv} x y with σπ-eq ty tv
...| no _ = []
diffμ*-mod x y
  | yes refl
  = skel <$> (diff1* x y >>= Patch-mapM (uncurry diffμ*-atoms))
```

Atoms are easy to diff. If reach two type variables, we tie the knot and keep patching. If we reach two constants, we set one into the other. Any other situation is forbiden.

```
diffμ*-atoms : {ty tv : Atom} → ⟦ ty ⟧_a → ⟦ tv ⟧_a → List (UU→AA Patchμ ty tv)
diffμ*-atoms {I ty} {I tv} x y   = fix <$> diffμ* x y
diffμ*-atoms {K ty} {K tv} x y   = return (set (→α {K ty} x , →α {K tv} y))
diffμ*-atoms {K ty} {I tv} x y   = []
diffμ*-atoms {I ty} {K tv} x y   = []
```

Insertions and deletions are very simple:

```
diffμ*-ins : {ty : U}{k : Fam_i} → Fix fam k → ⟦ ty ⟧ → List (Patchμ (T k) ty)
diffμ*-ins x y with sop y
diffμ*-ins x _ | strip cy dy
  = ins cy <$> alignμ' (x , unit) dy
```

```
diffμ*-del : {ty : U}{k : Fam_i} → ⟦ ty ⟧ → Fix fam k → List (Patchμ ty (T k))
diffμ*-del x y with sop x
diffμ*-del _ y | strip cx dx
  = del cx <$> alignμ' dx (y , unit)
```

We just need to pay attention not to insert or delete some constructor without arguments; This is done on the auxiliar alignment function:

```
alignμ'  : {ty tv : Π} → ⟦ ty ⟧p → ⟦ tv ⟧p
         → List (Al (UU→AA Patchμ) ty tv)
alignμ' {[]}       {_}        _ _  = []
alignμ' {_}        {[]}       _ _  = []
alignμ' {_ :: _} {_ :: _}  x y  = alignμ x y
```

Which ignores empty products. If we are not aligning empty products, we can just piggyback on the previous alignment function we had:

```
alignμ  : {ty tv : Π} → ⟦ ty ⟧p → ⟦ tv ⟧p
        → List (Al (UU→AA Patchμ) ty tv)
alignμ x y = align* x y >>= Al-mapM (uncurry diffμ*-atoms)
```

Now we just need to choose the patch with the least cost for the *deterministic* version.

```
diffμ : {k k' : Fam_i} → Fix fam k → Fix fam k' → Patchμ (T k) (T k')
diffμ {k} x y with diffμ* x y
...| s :: ss  = s <μ> ss
...| []       = set (unmu x , unmu y)
```

Needs discussion:

If we define a family of two types, say K ℕ::K Bool::[]. Now take x = inject fz *10* and y = inject (fs fz) true.

Here, diffmu* x y = []. It is easy to see why. We can't modify because we havetwo different types. We can't insert or delete because *both* inhabitants were constructed with constructors that have no recursive arguments.

In a real scenario, though, this will never happen. Mainly because we are only interested in diffing things of the same type. Sure, the types might change during the algorithm, but they start the same; hence at least a modify is always possible.

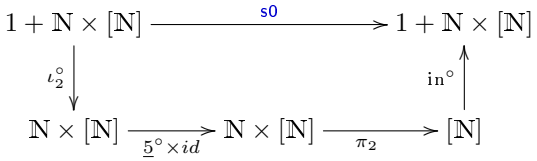
On another note; The aforementioned type family is not really a type family, but a single type: $\mathbb{N} + \mathbb{B}$.

Obviously, we can also define the "application" relation for fixed points, and that is done in `RegDiff/Diff/Multirec/Domain`. I believe it is more worthwhile to look at some example patches and their "application" relation instead of the general case, though. Let's begin with the lists we just discussed:

```
s0 : Patchμ LIST-F LIST-F
s0 = diffμ (5 > 8 > 13 > 21 > #) (8 > 13 > 21 > #)

s0-norm : Patchμ LIST-F LIST-F
s0-norm = del cons' (Ap1 5 (AX (fix (skel Scp)) A0))
```

Here, LIST-F is defined as u1⊕ℕ⊗I, the usual list functor. The "application" relation for the above patch is (isomorphic to):



4.1 Examples

Here we add some more examples of patches over fixpoints. These can be seen in the respective `Lab.agda` modules. Here are a few examples of list patches:

```
l0 l1 : list
l0 = (3 > 50 > 4 > #)
l1 = (1 > 50 > 4 > 20 > #)

s1 : Patchμ LIST-F LIST-F
s1 = diffμ l0 l1

s1-normalized : Patchμ LIST-F LIST-F
s1-normalized
  = skel
    (Scns cons'
      (CX fz fz (AX (set (→α 3 , →α 1)) A0) ::
        (CX fz fz
          (AX
            (fix
              (skel
                (Scns cons'
                  (CX fz fz (AX (set (→α 50 , →α 50)) A0) ::
                    (CX fz fz
                      (AX
                        (fix
                          (skel
                            (Scns cons'
                              (CX fz fz (AX (set (→α 4 , →α 4)) A0) ::
                                (CX fz fz
                                  (AX (fix (ins cons' (Ap1° 20 (AX (fix (skel Scp)) A0)))) A0)
                                  :: [])))))
                        A0)
                      :: [])))))
            A0)
          :: [])))
```

Previously we had *2-3-Trees* as an example. Here are some patches over them:

```
import RegDiff.Diff.Multirec.Base konstants keqs
  as DIFF
open DIFF.Internal (2-3-TREE-F :: []) public

k0 k1 k2 : 2-3-Tree
k0 = Leaf
k1 = 2-Node 1 Leaf Leaf
k2 = 3-Node 5 Leaf Leaf Leaf
k3 = 3-Node 3 k1 k2 k2

t1 t2 : 2-3-Tree
t1 = 2-Node 4 k1 k2
t2 = 3-Node 5 k1 Leaf k2
```

The patches we calculate are:

```
r1 r2 : Patchμ 2-3-TREE-F 2-3-TREE-F
r1 = diffμ t1 t2
r2 = diffμ k1 k3
```

Which are normalized to the following patches. Note that it is the A⊗ in r1 that lets

us copy k1 and k2 from the 2-node to the 3-node.

```
r1-normalized : Patchμ 2-3-TREE-F  2-3-TREE-F
r1-normalized
  = skel
    (SX
      (CX 2-node' 3-node'
        (AX (set (i1 (4 , unit) , i1 (5 , unit)))
          (AX (fix (skel Scp))
            (Ap1° ⟨ i1 unit ⟩ (AX (fix (skel Scp)) A0))))))

r2-normalized : Patchμ 2-3-TREE-F 2-3-TREE-F
r2-normalized
  = ins 3-node'
    (Ap1° 3
      (AX (fix (skel Scp))
        (Ap1°
          ⟨ i2 (i2 (i1 (5 , ⟨ i1 unit ⟩ , ⟨ i1 unit ⟩ , ⟨ i1 unit ⟩ , unit)))
          ⟩
          (Ap1°
            ⟨ i2 (i2 (i1 (5 , ⟨ i1 unit ⟩ , ⟨ i1 unit ⟩ , ⟨ i1 unit ⟩ , unit)))
            ⟩
            A0))))
```

5 Comparing to the Rose Tree approach

Needs discussion:

- By using a list of edit operations, they lose the alignments.
- They allow for sharing of data even when the same constructor is used.