Diffing Mutually Recursive Types

A code tour

Victor Cacciari Miraldo
University of Utrecht

December 9, 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 have a different *type* than the codes for types. This requires more discipline to organize our code. 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 → 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 : VecI 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 a 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 $img\ R \subseteq id$) 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 $(\text{inv } p)$ 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:

$$\begin{tikzcd}
B \arrow[r, "apply_{(\text{inv } p)}"] \arrow[rrd, "\iota_1"'] & A + 1 \arrow[r, "apply_p + id"] & (B+1)+1 \arrow[d, "\mu"'] \\
 & & B + 1
\end{tikzcd}$$

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, $(\text{inv } p)$ 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.

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 PF style:

$$\begin{tikzcd}
\llbracket ty \rrbracket_A \arrow[rr, bend left, "\underline{y} \cdot \underline{x}^\circ"] & K \arrow[l, "\underline{x}"] \arrow[r, "\underline{y}"'] & \llbracket tv \rrbracket_A
\end{tikzcd}$$

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 Δ 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 $\Delta_x = \mathsf{delta}\ [\![\cdot]\!]_{\mathsf{x}}$, 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, the 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 Δ 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.

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)
        → ListI (contr P ∘ α) (typeOf ty i)
        → S P ty
```

Remember that $\mathsf{contr}\ P\ x = P\ x\ x$ and $\alpha : \mathsf{Atom}\ n \rightarrow \sigma\pi\ n$; Here, $\mathsf{ListI}\ P\ l$ 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 (mapᵢ 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.

```
spine-cns : {ty : U}(x y : ⟦ ty ⟧) → S Δ_s 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 (zip_p dx dy)

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

zip_p : {ty : Π}
   → ⟦ ty ⟧_p → ⟦ ty ⟧_p → ListI (λ k → Δ_s (α k) (α k)) ty
zip_p {[]} _ _
   = []
zip_p {_ ∷ ty} (x , xs) (y , ys)
   = (i1 (x , unit) , i1 (y , unit)) ∷ zip_p xs ys
```

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

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

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

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

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 (SX p) case, we simply ask for the “application” relation of p. The algorithm produces a S Δ_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 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 ⟦2-3-TREE-F⟧$_{\mathsf{Unit}}$ 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.

$$\begin{tikzcd}
V & T \arrow[l, "(\mathsf{CX}\ i\ j\ p)^\flat"'] \arrow[d, "\mathsf{inj}_i{}^\circ"] \\
\mathsf{typeOf}\ V\ j \arrow[u, "\mathsf{inj}_j"] & \mathsf{typeOf}\ T\ i \arrow[l, "p^\flat"]
\end{tikzcd}$$

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 $\{-\} \not\subseteq \mathcal{L}$ 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.

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 out search-space.

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.

```
a = C-mapM align* c
  = SX (CX 2-node' 3-node' (AX (10 , 10) (AX (unit , unit) ···)))
   ::SX (CX 2-node' 3-node' (Ap1 10 (AX (unit , 10) ···)))
   ::SX (CX 2-node' 3-node' (Ap1 10 (Ap1 unit ···)))
   ::SX (CX 2-node' 3-node' (Ap1° 10 (AX (10 , unit) ···)))
    ...
```

Now we have a problem. Which of the patches above should we chose to be *the* patch? Recall that we mentioned 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:

$$\begin{tikzcd}
(\mathsf{AX}\ a_1\ a_2)^\flat = B \times \Pi D & A \times \Pi C \arrow[l, "{a_1}^\flat \times {a_2}^\flat"'] \\
(\mathsf{Ap1}\ x\ a)^\flat = \Pi B & X \times \Pi A \arrow[l, "\langle \underline{x}, {a^\flat}^\circ \rangle^\circ"'] \\
(\mathsf{Ap1}^\circ\ x\ a)^\flat = X \times \Pi B & \Pi A \arrow[l, "\langle \underline{x}, a^\flat \rangle"'] \\
\mathsf{A0}^\flat = \mathbb{1} & \mathbb{1} \arrow[l, "\top"']
\end{tikzcd}$$

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:

$$\begin{aligned}\mathsf{Type1}\ &= \mathsf{K}\ \mathsf{k}\mathbb{N} \otimes []\\ &\oplus \mathsf{K}\ \mathsf{k}\mathbb{N} \otimes \mathsf{K}\ \mathsf{k}\mathbb{N} \otimes []\\ &\oplus []\end{aligned}$$

It clearly has two constructors:

$$\begin{aligned}\mathsf{C}_1\ \mathsf{C}_2\ &: \mathsf{Constr}\ \mathsf{Type1}\\ \mathsf{C}_1\ &= \mathsf{fz}\\ \mathsf{C}_2\ &= \mathsf{fs}\ \mathsf{fz}\end{aligned}$$

Now, let's take two inhabitants of Type1.

$$\begin{aligned}\mathsf{x}\ \mathsf{y}\ &: [\![\ \mathsf{Type1}\ ]\!]\\ \mathsf{x}\ &= \mathsf{inject}\ \mathsf{C}_2\ (\mathsf{4}\ ,\ \mathsf{10}\ ,\ \mathsf{unit})\\ \mathsf{y}\ &= \mathsf{inject}\ \mathsf{C}_1\ (\mathsf{10}\ ,\ \mathsf{unit})\end{aligned}$$

There are two possible options for diff $x\ y$:

$$\begin{aligned}&\mathsf{ds} : \mathsf{Patch^*}\ \mathsf{Type1} && && && .\\ &\mathsf{ds}\ = \mathsf{SX}\ (\mathsf{CX}\ \mathsf{C}_2\ \mathsf{C}_1\ (\mathsf{AX} && (\mathsf{4}\ ,\ \mathsf{10})\ \ (\mathsf{Ap1}\ \ \mathsf{10} && && \mathsf{A0})))\ \texttt{-}\ \texttt{P1}\\ &\quad ::\ \mathsf{SX}\ (\mathsf{CX}\ \mathsf{C}_2\ \mathsf{C}_1\ (\mathsf{Ap1}\ \ \mathsf{4} && && (\mathsf{AX}\ \ (\mathsf{10}\ ,\ \mathsf{10}) && \mathsf{A0})))\ \texttt{-}\ \texttt{P2}\\ &\quad ::\ []\end{aligned}$$

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:

$$\begin{aligned}P1^\flat &= \mathsf{inj}_{\mathsf{C}_1}\ \cdot\ \langle(\underline{4}\ \cdot\ \underline{10}^\circ), \underline{10}\rangle^\circ\ \cdot\ \mathsf{inj}_{\mathsf{C}_2}{}^\circ\\ P2^\flat &= \mathsf{inj}_{\mathsf{C}_1}\ \cdot\ \langle\underline{4}, id\rangle^\circ\ \cdot\ \mathsf{inj}_{\mathsf{C}_2}{}^\circ\end{aligned}$$

Drawing them in a diagram we have:

$$\begin{tikzcd}
\mathsf{typeOf}_{\mathsf{Type1}}\ \mathsf{C}_2 \equiv \mathbb{N}\times\mathbb{N} \arrow[d, shift right, "<\underline{4},id>^\circ"'] \arrow[d, shift left, "<\underline{4}\cdot\underline{10}^\circ,\underline{10}>^\circ"] & & [\![\mathsf{Type1}]\!] \arrow[ll, "\mathsf{inj}_{\mathsf{C}_2}{}^\circ"'] \arrow[d, dashed, shift right, "P2^\flat"'] \arrow[d, dashed, shift left, "P1^\flat"] \\
\mathsf{typeOf}_{\mathsf{Type1}}\ \mathsf{C}_1 \equiv \mathbb{N} \arrow[rr, "\mathsf{inj}_{\mathsf{C}_1}"'] & & [\![\mathsf{Type1}]\!]
\end{tikzcd}$$

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:

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

The first proof obligation is easy to calculate with:

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

The second is easier to prove once we add variables!

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

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

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.

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