Diffing Mutually Recursive Types

A code tour

Victor Cacciari Miraldo
University of Utrecht

November 24, 2016

1 Our Universe

The universe we are using is a variant of Regular types, but instead of having only one type variable, we handle n type variables. The codes are description of regular functors on n variables:

```
data U_n (n : ℕ) : Set where
  I   : Fin n          → U_n n
  K   : Fin ks#        → U_n n
  u1  :                  U_n n
  _⊕_ : (ty tv : U_n n) → U_n n
  _⊗_ : (ty tv : U_n n) → U_n 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. The denotation is defined as:

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

⟦_⟧ : {n : ℕ} → U_n n → Parms n → Set
⟦ I x     ⟧ A = A x
⟦ K x     ⟧ A = lookup x ks
⟦ u1      ⟧ A = Unit
⟦ ty ⊕ tv ⟧ A = ⟦ ty ⟧ A ⊎ ⟦ tv ⟧ A
⟦ ty ⊗ tv ⟧ A = ⟦ ty ⟧ A × ⟦ tv ⟧ 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 (U_n 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 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)
```

I still have no good justification for the *parm-size* field. Later on I sketch what I believe is the real meaning of the cost function.

The following sections discuss functionality that does not depend on *parameters to codes*. We will be passing them as Agda module parameters. 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*.

We then declare the following synonyms:

```
U : Set
U = Uₙ 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

UUSet : Set₁
UUSet = U → U → Set
```

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 $apply_p\ x \equiv$ just y.

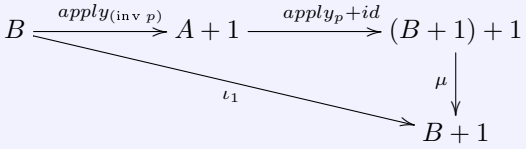
Well, $apply_p$ can be seen as a functional relation (R is functional iff $img\ R \subseteq id$) from A to B. We call this the "application" relation of the patch, and we will denote it by $p^\flat \subseteq A \times B$.

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:



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.

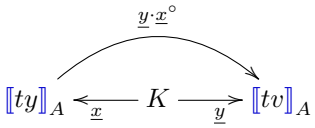
Now, let us discuss some code and build some intuition for what is what in the above schema. We will present different parts of the code, how do they relate to this relational view and give examples here and there!

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 just fine.

```
Δ : UUSet
Δ ty tv = ⟦ ty ⟧ A × ⟦ tv ⟧ A
```

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

2.2 Spines

We can try to make it better by identifying the longest prefix of constructors where x and y agree, before giving up and using Δ. Moreover, this becomes much easier if x and y actually have 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 the longest common prefix of x and y:

```
data S (P : UUSet) : U → Set where
  SX  : {ty : U} → P ty ty → S P ty
  Scp : {ty : U} → S P ty
  S⊗  : {ty tv : U}
      → S P ty → S P tv → S P (ty ⊗ tv)
  Si1 : {ty tv : U}
      → S P ty → S P (ty ⊕ tv)
  Si2 : {ty tv : U}
      → S P ty → S P (tv ⊕ ty)
```

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}
      → (f : ∀{k} → P k k → Q k k)
      → S P ty → S Q ty
S-map f (SX x)     = SX (f x)
S-map f Scp        = Scp
S-map f (S⊗ s o)   = S⊗ (S-map f s) (S-map f o)
S-map f (Si1 s)    = Si1 (S-map f s)
S-map f (Si2 s)    = Si2 (S-map f s)
```

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 inspect the first constructor and traverse it. Then we repeat.

```
mutual
  spine-cp : {ty : U} → ⟦ ty ⟧ A → ⟦ ty ⟧ A → S Δ ty
  spine-cp {ty} x y
    with dec-eq _≟-A_ ty x y
  ...| no  _ = spine x y
  ...| yes _ = Scp

  spine : {ty : U} → ⟦ ty ⟧ A → ⟦ ty ⟧ A → S Δ ty
  spine {ty ⊗ tv} (x1 , x2) (y1 , y2)
    = S⊗ (spine-cp x1 y1) (spine-cp x2 y2)
  spine {tv ⊕ tw} (i1 x)    (i1 y)  = Si1 (spine-cp x y)
  spine {tv ⊕ tw} (i2 x)    (i2 y)  = Si2 (spine-cp x y)
  spine {ty}      x         y       = SX (delta {ty} {ty} x y)
```

The "application" relations specified by a spine s = spine-cp x y, denoted $s^\flat$ are:

$$\begin{aligned}
\mathsf{Scp}^\flat &= A \xleftarrow{id} A \\
(\mathsf{S}\otimes\ s_1\ s_2)^\flat &= A \times B \xleftarrow{s_1^\flat \times s_2^\flat} A \times B \\
(\mathsf{Si1}\ s)^\flat &= A + B \xleftarrow{\iota_1 \cdot s^\flat \cdot \iota_1^\circ} A + B \\
(\mathsf{Si2}\ s)^\flat &= A + B \xleftarrow{\iota_2 \cdot s^\flat \cdot \iota_2^\circ} A + B \\
(\mathsf{SX}\ p)^\flat &= A \xleftarrow{p^\flat} A
\end{aligned}$$

Note that, in the (SX p) case, we simply ask for the "application" relation of p.

Needs discussion:

Non-cannonicity can be a problem: $\mathsf{Scp}^\flat \equiv (\mathsf{S}\otimes\ \mathsf{Scp}\ \mathsf{Scp})^\flat$ Even though the spine-cp function will never find the right-hand above, it feels sub-optimal to allow this.
One possible solution could be to remove Scp and handle them through the maybe monad. Instead of S Δ we would have S (Maybe Δ), where the nothings represent copy. This ensures that we can only copy on the leaves. Branch `explicit-cpy` of the repo has this experiment going. It is easier said than done.

Ignoring the problems and moving forward; note that for any x and y, a spine s = spine-cp x y will NEVER contain a product nor a unit on a leaf (we force going through products and copying units). Hence, whenever we are traversing s and find a SX, we know that: (1) the values of the pair are different and (2) we must be at a coproduct, a constant type or a type variable. The constant type or the type variable are out of our control. But we can refine our *description* in case we arrive at a coproduct.

2.3 Coproduct Changes

Let's imagine we are diffing the following values of a type $\mathbb{1} + (\mathbb{N}^2 + \mathbb{N}) \times \mathbb{N}$:

$$\begin{aligned}
s &= \mathsf{spine\text{-}cp}\ (\iota_2\ (\iota_1\ (4\ ,\ 10)\ ,\ 5))\ (\iota_2\ (\iota_2\ 10\ ,\ 5)) \\
&= \mathsf{Si2}\ (\mathsf{S}\otimes\ (\mathsf{SX}\ (\iota_1\ (4\ ,\ 10)\ ,\ \iota_2\ 10))\ \mathsf{Scp})
\end{aligned}$$

And now, we want to S-map over s and refine the Δ inside. We want to have information about which injections we need to pattern match on and which we need to introduce. We begin with a type (that's also a free monad) that encodes the injections we need to insert or pattern-match on:

```
data C (P : UUSet) : U → U → Set where
  CX    : {ty tv : U}  → P ty tv → C P ty tv
  Ci1   : {ty tv k : U} → C P ty tv → C P ty (tv ⊕ k)
  Ci2   : {ty tv k : U} → C P ty tv → C P ty (k ⊕ tv)
  Ci1°  : {ty tv k : U} → C P ty tv → C P (ty ⊕ k) tv
  Ci2°  : {ty tv k : U} → C P ty tv → C P (k ⊕ ty) tv
```

Note that C is also a functor on P:

```
C-map : {ty tv : U}{P Q : UUSet}
  → (f : ∀{k v} → P k v → Q k v)
  → C P ty tv → C Q ty tv
C-map f (CX x)   = CX (f x)
C-map f (Ci1 s)  = Ci1 (C-map f s)
C-map f (Ci2 s)  = Ci2 (C-map f s)
C-map f (Ci1° s) = Ci1° (C-map f s)
C-map f (Ci2° s) = Ci2° (C-map f s)
```

And we make an eager function that will consume ALL coproduct injections from both source and destination:

$$
\begin{aligned}
&\mathsf{change} : \{ty\ tv : \mathsf{U}\} \to [\![\ ty\ ]\!]\ A \to [\![\ tv\ ]\!]\ A \to \mathsf{C}\ \Delta\ ty\ tv \\
&\mathsf{change}\ \{ty\}\ \{tv \oplus tw\}\ x\ (\mathsf{i1}\ y) = \mathsf{Ci1}\ \ (\mathsf{change}\ x\ y) \\
&\mathsf{change}\ \{ty\}\ \{tv \oplus tw\}\ x\ (\mathsf{i2}\ y) = \mathsf{Ci2}\ \ (\mathsf{change}\ x\ y) \\
&\mathsf{change}\ \{ty \oplus tw\}\ \{tv\}\ (\mathsf{i1}\ x)\ y = \mathsf{Ci1}^\circ\ (\mathsf{change}\ x\ y) \\
&\mathsf{change}\ \{ty \oplus tw\}\ \{tv\}\ (\mathsf{i2}\ x)\ y = \mathsf{Ci2}^\circ\ (\mathsf{change}\ x\ y) \\
&\mathsf{change}\ \{ty\}\ \{tv\}\ \ x\ \ y = \mathsf{CX}\ (\mathsf{delta}\ \{ty\}\ \{tv\}\ x\ y)
\end{aligned}
$$

Now, we could S-map change over s:

$$
\begin{aligned}
\mathsf{S\text{-}map}\ \mathsf{change}\ s &= \mathsf{Si2}\ (\mathsf{S}\otimes\ (\mathsf{SX}\ (\mathsf{Ci2}\ (\mathsf{Ci1}^\circ\ (\mathsf{CX}\ ((4\ ,\ 10)\ ,\ 10)))))\ \mathsf{Scp}) \\
&= s'
\end{aligned}
$$

The whole thing also becomes of a much more expressive type:
$s' = \mathsf{S\text{-}map}\ \mathsf{change}\ s\ :\ \mathsf{S}\ (\mathsf{C}\ \Delta)$

We can read the type as: a common prefix from both terms followed by injections into the target term and pattern matching on the source term followed by point-wise changes from source to destination.

Note that we can also say, for sure, that we will never have a $\mathsf{SX}\ (\mathsf{Ci1}\ (\mathsf{Ci1}^\circ\ c))$ at the S-leaves of s'; this would mean that we need to inject the target with ι_1 and pattern match the source on ι_1, but this is a common-prefix! Hence it would be recognized by spine-cp and instead we would have: $\mathsf{Si1}\ (\mathsf{SX}\ c)$.

Needs discussion:

Even though the algorithm does not produce $\mathsf{SX}\ (\mathsf{Ci1}\ (\mathsf{Ci1}^\circ\ c))$, as mentioned above, this is still a valid inhabitant of $\mathsf{S}\ (\mathsf{C}\ \Delta)$!

Just like the values of S, we can also define the "application" relation induced by the values of C:

$$
\begin{aligned}
(\mathsf{Ci1}\ c)^\flat &= B + X \xleftarrow{\iota_1 \cdot c^\flat} A \\
(\mathsf{Ci2}\ c)^\flat &= X + B \xleftarrow{\iota_2 \cdot c^\flat} A \\
(\mathsf{Ci1}^\circ\ c)^\flat &= B \xleftarrow{c^\flat \cdot \iota_1^\circ} A + X \\
(\mathsf{Ci2}^\circ\ c)^\flat &= B \xleftarrow{c^\flat \cdot \iota_2^\circ} X + A \\
(\mathsf{CX}\ p)^\flat &= B \xleftarrow{p^\flat} A
\end{aligned}
$$

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

2.4 Aligning Everything

Following a similar reasoning as from S to C; the leaves of a C produced through change will NEVER contain a coproduct as the topmost type. Hence, we know that they will contain either a product, or a constant type, or a type variable. In the case of a constant type or a type variable, there is not much we can do at the moment, but for a product we can refine this a little bit more before using Δ[1].

Looking at our running example, we have a leaf $(\mathsf{CX}\ ((4\ ,\ 10)\ ,\ 10))$ to take care of. Here the source type is $\mathbb{N}^2$ and the destination is $\mathbb{N}$. Note that the $\mathbb{N}$ is treated as

[1] In fact, splitting the different stages of the algorithm into different types reinforced our intuition that the alignment is the source of difficulties. As we shall see, we now need to introduce non-determinism.

a constant type here. As we mentioned above, we have a product on the source, so we could extract some more information before giving up and using Δ!

Here is where our design space starts to be huge. Our definition of alignment is:

```
data Al (P : UUSet) : U → U → Set where
  AX   : {ty tv : U}      → P ty tv → Al P ty tv
  Ap1  : {ty tv tw : U}   → ⟦ tw ⟧ A → Al P ty tv → Al P ty (tv ⊗ tw)
  Ap1° : {ty tv tw : U}   → ⟦ tw ⟧ A → Al P ty tv → Al P (ty ⊗ tw) tv
  Ap2  : {ty tv tw : U}   → ⟦ tw ⟧ A → Al P ty tv → Al P ty (tw ⊗ tv)
  Ap2° : {ty tv tw : U}   → ⟦ tw ⟧ A → Al P ty tv → Al P (tw ⊗ ty) tv
  A⊗   : {ty tv tw tz : U}
       → Al P ty tw → Al P tv tz → Al P (ty ⊗ tv) (tw ⊗ tz)
```

Which states that we can force components of the source or destination product to be equal to a given value or we can join two alignments together (This is a big source of inefficiency!).

Computing alignments is very expensive, specially if done in the dumb way. We distinguish the expensive align function with an *exp* suffix. In the case we actually have products on both the source and the destination we have a lot of options (hence the list monad!):

```
align-exp : {ty tv : U} → ⟦ ty ⟧ A → ⟦ tv ⟧ A → List (Al Δ ty tv)
align-exp {ty ⊗ ty'} {tv ⊗ tv'} (x1 , x2) (y1 , y2)
  = A⊗ <$> align-exp x1 y1 <*> align-exp x2 y2
  ++ Ap1  y2 <$> align-exp (x1 , x2) y1
  ++ Ap2  y1 <$> align-exp (x1 , x2) y2
  ++ Ap1° x2 <$> align-exp x1 (y1 , y2)
  ++ Ap2° x1 <$> align-exp x2 (y1 , y2)
```

If we only have products on the left or on the right (or none), we have less options:

```
align-exp {ty ⊗ ty'} {tv} (x1 , x2) y
  = Ap1° x2 <$> align-exp x1 y
  ++ Ap2° x1 <$> align-exp x2 y
align-exp {ty} {tv ⊗ tv'} x (y1 , y2)
  = Ap1  y2 <$> align-exp x y1
  ++ Ap2  y1 <$> align-exp x y2
align-exp {ty} {tv} x y = return (AX (x , y))
```

Following our previous example, we could C-mapM our alignment function (the C-mapM is the monadic variant of C-map) on s', in order to find all alignments of $(4, 10)$ and 10. In this case, this is very easy[2].

```
C-map align s' =[ Si2 (S⊗ (SX (Ci2 (Ci1° (CX (Ap1° 10 (AX (4 , 10)))))) Scp)
                , Si2 (S⊗ (SX (Ci2 (Ci1° (CX (Ap2° 4 (AX (10 , 10)))))) Scp) ]
```

If we ommit the SX and CX constructors this becomes slightly more readable:

```
C-map align s' =[ Si2 (S⊗ (Ci2 (Ci1° (Ap1° 10 (AX (4 , 10))))) Scp)
                , Si2 (S⊗ (Ci2 (Ci1° (Ap2° 4 (AX (10 , 10))))) Scp) ]
```

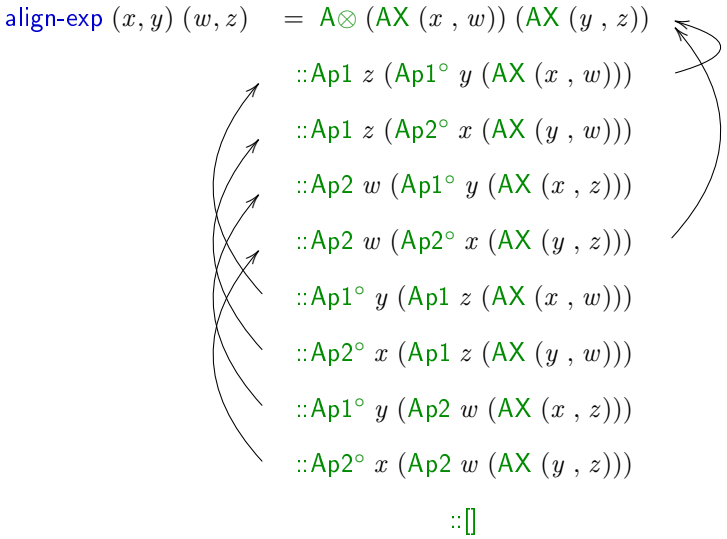
But now we end up having to choose between one of those to be *the* patch. This is where we start to need a cost function. Before talking about cost, let's look at the "application" relations of Al:

[2] It might be hard to build intuition for why we need the A⊗ constructor. On the `Lab` module of Fixpoint there is an example using 2-3-Trees that motivates the importance of that constructor

$$
\begin{aligned}
(\mathsf{A}\otimes\ a_1\ a_2)^\flat &= B \times D \xleftarrow{a_1^\flat \times a_2^\flat} A \times C \\
(\mathsf{Ap1}\ x\ c)^\flat &= B \times X \xleftarrow{\langle c^\flat, \underline{x}\rangle} A \\
(\mathsf{Ap2}\ x\ c)^\flat &= X \times B \xleftarrow{\langle \underline{x}, c^\flat\rangle} A \\
(\mathsf{Ap1}^\circ\ x\ c)^\flat &= B \xleftarrow{\pi_1 \cdot (c^\flat \times \underline{x}^\circ)} A \times X \\
(\mathsf{Ap2}^\circ\ x\ c)^\flat &= B \xleftarrow{\pi_2 \cdot (\underline{x}^\circ \times c^\flat)} X \times A \\
(\mathsf{AX}\ p)^\flat) &= B \xleftarrow{p^\flat} A
\end{aligned}
$$

2.5 A simple optimization

Looking carefully at the align-exp function above, we are computing a lot of unecessary branches, specially in the case for products on both sides. Below we compute align-exp $(x\ ,\ y)\ (w\ ,\ z)$ and put arrows indicating which AI have an isomorphic underlying "application" relation.



If we then look at which AIgnments do *not* have an arrow out of it, that is, the ones that don't have an isomorphic alignment also being computed, we get:

```
align (x , y) (w , z) = A⊗ (AX (x , w)) (AX (y , z))
                        ::Ap1 z (Ap2° x (AX (y , w)))
                        ::Ap2 w (Ap1° y (AX (x , z)))
                        ::[]
```

We hence simplify the alignment function:

```
align : {ty tv : U} → ⟦ ty ⟧ A → ⟦ tv ⟧ A → List (AI Δ ty tv)
align {ty ⊗ ty'} {tv ⊗ tv'} (x1 , x2) (y1 , y2)
  = A⊗ <$> align x1 y1 <*> align x2 y2
  ++ (Ap1  y2 ∘ Ap2° x1) <$> kill-p2 (align x2 y1)
  ++ (Ap2  y1 ∘ Ap1° x2) <$> kill-p1 (align x1 y2)
```

```
align {ty ⊗ ty'} {tv} (x1 , x2) y
  = Ap1° x2 <$> align x1 y
  ++ Ap2° x1 <$> align x2 y
align {ty} {tv ⊗ tv'} x (y1 , y2)
  = Ap1  y2 <$> align x y1
  ++ Ap2  y1 <$> align x y2
align {ty} {tv} x y = return (AX (x , y))
```

The kill-p2 (resp. kill-p1) functions above are used to prune the search space further. They will ignore every branch that starts with a Ap2 (resp Ap1), which will, for sure, have had an isomorphic alignment computed already (in terms of A⊗). Writing down the "application" relations makes this easy to see.

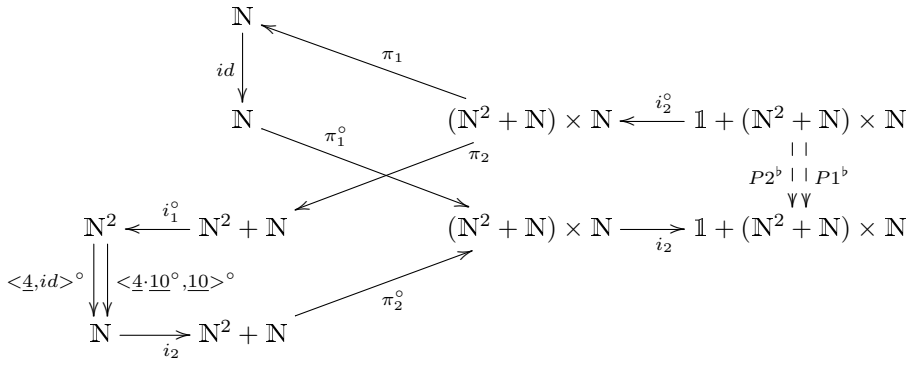
3 Patches as Relations

Here we will be using the semantics for $(x\ ,\ y)^\flat$ as described in the discussion box at Section 2.1. That is, we would like to say that the patch (P2), above, copies the 10, instead of saying that 10 changes into 10. We can postpone this by changing the relation semantics of Δ. Hence: $(x,x)^{\flat\cdot} = id$ and $(x,y)^\flat = \underline{y} \cdot \underline{x}^\circ$.

Nevertheless, if we look at the two patches above as relations, we have:

$$P1^\flat = i_2 \cdot (i_2 \cdot < \underline{4} \cdot \underline{10}^\circ, \underline{10} >^\circ \cdot i_1^\circ \times id) \cdot i_2^\circ$$
$$P2^\flat = i_2 \cdot (i_2 \cdot < \underline{4}, id >^\circ \cdot i_1^\circ \times id) \cdot i_2^\circ$$

Writing them in a diagram:



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:

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

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.

This suggests an interesting justification for the cost function. For some reason, looks like we won the lottery with our cost functions. We are always choosing the patch that gives rise to the maximal relation. I still don't clearly understand why or how, but it works.

TODO

Below are our cost functions:
we should actually be able to choose patches without cost functions... discuss this up! Use the align optimization example...
ADD VALUES, FORGET ABOUT THOSE RAW FUNCTIONS!

```
cost-Δ : {ty tv : U} → Δ ty tv → ℕ
cost-Δ {ty} {tv}  (x , y) with U-eq ty tv
cost-Δ {ty} {.ty} (x , y) | yes refl
   with dec-eq _≟-A_ ty x y
...| yes _ = 0
...| no  _ = cost-Δ-raw {ty} {ty} (x , y)
cost-Δ {ty} {tv}  (x , y) | no _
   = cost-Δ-raw {ty} {tv} (x ,  y)

S-cost : {ty : U}{P : UUSet}
   → (costP : ∀{ty} → P ty ty → ℕ)
   → S P ty → ℕ
S-cost c (SX x)  = c x
S-cost c Scp     = 0
S-cost c (S⊗ s o) = S-cost c s + S-cost c o
S-cost c (Si1 s)  = S-cost c s
S-cost c (Si2 s)  = S-cost c s

C-cost : {ty tv : U}{P : UUSet}
   → (costP : ∀{ty tv} → P ty tv → ℕ)
   → C P ty tv → ℕ
C-cost c (CX x)   = c x
C-cost c (Ci1 s)  = C-cost c s
C-cost c (Ci2 s)  = C-cost c s
C-cost c (Ci1° s) = C-cost c s
C-cost c (Ci2° s) = C-cost c s

Al-cost : {ty tv : U}{P : UUSet}
   → (costP : ∀{ty tv} → P ty tv → ℕ)
   → Al P ty tv → ℕ
Al-cost c (AX xy) = c xy
Al-cost c (A⊗ s o) = Al-cost c s + Al-cost c o
Al-cost c (Ap1  {tw = k} x s) = Al-cost-raw {k} x (Al-cost c s)
Al-cost c (Ap2  {tw = k} x s) = Al-cost-raw {k} x (Al-cost c s)
Al-cost c (Ap1° {tw = k} x s) = Al-cost-raw {k} x (Al-cost c s)
Al-cost c (Ap2° {tw = k} x s) = Al-cost-raw {k} x (Al-cost c s)
```

3.1 Patches for Regular Types

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

```
Patch : U → Set
Patch ty = S (C (Al Δ)) ty
```

Computing inhabitants of such type is done with:

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

4 Mutually Recursive Types

TODO

Re-explain why we removed the skel constructor.

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 (U_n 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} → (U_n n → U_n n → Set) → (U_n n → U_n n → Set) → (U_n n → U_n n → Set)
(P +ᵤ Q) ty tv = (P ty tv) ⊎ (Q ty tv)
```

Now, we already have the ingredients for common prefixes, coproducts and products. We now need to handle type variables. Before we proceed with the nasty definitions, we still need two last synonyms:

```
Fam_i : Set
Fam_i = Fin fam#

T : Fam_i → U_n fam#
T k = lookup k fam
```

Here T k represents the k-th type of the family, and Fam$_i$ represents the indexes of our family.

Remember that we have Spines, to handle the common prefixes; Changes to handle different coproduct injections on source and destination and Alignments to make sure we exploit products. We kept mentioning that we could not do anything on the leaves that were type variables or constant types. Well, constant types we still can't, and we have to stick to Δ, but now we need to handle the type-variables.

A patch for a fixed point might not 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!

Here is the definition:

```
data Patchμ : U → U → Set where
  chng : {ty tv  : U}      → Cμ Patchμ ty tv      → Patchμ ty tv
  fix  : {k k'   : Fam_i}  → Patchμ (T k) (T k')  → Patchμ (I k) (I k')
  set  : {ty     : U}      → Δ ty ty              → Patchμ ty ty
```

The skel constructor lets a spine kick in, which has Patchμ on it's leaves again. The fix constructor ties the know between type-variables and their lookup in the family, which is isomorphic. The set constructor specifies that we are setting something. The Cμ type extends the previous C with options for inserting and deleting:

```
data Cμ (P : UUSet) : U → U → Set where
  Cins  : {k k' : Fam_i} → C (Al P) (I k)  (T k')  → Cμ P (T k) (T k')
  Cdel  : {k k' : Fam_i} → C (Al P) (T k)  (I k')  → Cμ P (T k) (T k')
  Cmod  : {ty : U}       → S (C (Al P)) ty         → Cμ P ty     ty
```

Note that fix, Cins and Cdel are heterogeneous on the Fam_i indexes! This is very important for mutually recursive families.

Needs discussion:

I have been experimenting with different cost models and slight variations on the definitions of Patchμ. I found that requiring set to set the *same* type is a great idea! Not only it removes a lot of absurd patches, but it also speeds up the process quite a bit:
$set : \{ty : U\} \to \Delta\ ty\ ty \to Patch\mu\ ty\ ty$

TODO

Not anymore! discuss this!

Computing a Patchμ is done by piggybacking on the functions for computing S, C and Al separately, then mapping over them with some refinement functions:

```
changeμ : {ty tv : U}
  → ⟦ ty ⟧ (Fix fam) → ⟦ tv ⟧ (Fix fam)
  → List (C (Al Patchμ) ty tv)
changeμ x y = C-mapM ((_ >>= Al-mapM refine-Al) ∘ uncurry align) (change x y)

try-mod : {ty tv : U} → ⟦ ty ⟧ (Fix fam) → ⟦ tv ⟧ (Fix fam) → List (Patchμ ty tv)
try-mod {ty} {tv}  x y with U-eq ty tv
try-mod {ty} {tv}  x y | no _   = []
try-mod {ty} {.ty} x y | yes refl
  = (chng ∘ Cmod) <$> S-mapM (uncurry changeμ) (spine-cp x y)

{-# TERMINATING #-}
diffμ* : {k k' : Fam_i} → Fix fam k → Fix fam k' → List (Patchμ (T k) (T k'))
diffμ* {k} {k'} ⟨ x ⟩ ⟨ y ⟩
  =  try-mod {T k} {T k'} x y
  ++ ((chng ∘ Cdel {k = k} {k'}) <$> changeμ x ⟨ y ⟩)
  ++ ((chng ∘ Cins {k = k} {k'}) <$> changeμ ⟨ x ⟩ y)
```

The refinement functions are given by:

```
mutual
  refine-Al : {k v : U} → Δ k v → List (Patchμ k v)
  refine-Al {I k} {I k'} (x , y) = fix <$> diffμ* x y
  refine-Al {k}   {v}    (x , y) with U-eq k v
  refine-Al {k}   {v}    (x , y) | no _   = []
  refine-Al {k}   {.k}   (x , y) | yes refl = return (set (x , y))
```

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:

TODO

recompile the examples