Diffing Mutually Recursive Types

A code tour

Victor Cacciari Miraldo
University of Utrecht

November 17, 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
```

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 barebones 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 depent 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 throughtout 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 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 can be seen as it's "application" (partial) function. That is, a relation $e \subseteq A \times B$ such that $img\ e \subseteq id$ (e is functional).

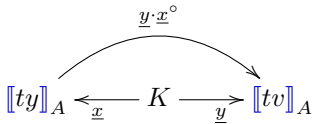
Now, let us discuss some code and build some intuition for what is what in the above schema.

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 acomplished 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 "apply" 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 indo y.

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. 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 "apply" relations specified by a spine s, denoted $s^\flat$ are:

$$
\begin{aligned}
\mathsf{Scp}^\flat &= A \xleftarrow{id} A \\
(\mathsf{S}\otimes\ s_1\ s_2)^\flat &= A \times B \xleftarrow{s_1^\flat \times s_2^\flat} A \times B \\
(\mathsf{Si1}\ s)^\flat &= A + B \xleftarrow{i_1 \cdot s^\flat \cdot i_1^\circ} A + B \\
(\mathsf{Si2}\ s)^\flat &= A + B \xleftarrow{i_2 \cdot s^\flat \cdot i_2^\circ} A + B \\
(\mathsf{SX}\ (x, y))^\flat &= A \xleftarrow{y \cdot x^\circ} A
\end{aligned}
$$

This has some problems that I do not like. Namelly:

- Non-cannonicity: $\mathsf{Scp}^\flat \equiv (\mathsf{S}\otimes\ \mathsf{Scp}\ \mathsf{Scp})^\flat$ 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 $s = \mathsf{spine\text{-}cp}\ x\ y$ 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 $\mathbb{1} + (\mathbb{N}^2 + \mathbb{N}) \times \mathbb{N}$:

$$
\begin{aligned}
s &= \mathsf{spine\text{-}cp}\ (\mathsf{i2}\ (\mathsf{i1}\ (4\ ,\ 10)\ ,\ 5))\ (\mathsf{i2}\ (\mathsf{i2}\ 10\ ,\ 5)) \\
&= \mathsf{Si2}\ (\mathsf{S}\otimes\ (\mathsf{SX}\ (\mathsf{i1}\ (4\ ,\ 10), \mathsf{i2}\ 10))\ \mathsf{Scp})
\end{aligned}
$$

And now, we want to S-map over s and refine the Δ inside to another type, that contains information about which injections we need to pattern match on and which we need to introduce. One step at a time, though. Let's first look how could we represent this information:

We begin with a type (that's also a free monad) that encodes the injections we need to insert on the *destination*:

```
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)
```

And we make an eager function that will consume ALL injections from the right:

```
change : {ty tv : U} → ⟦ ty ⟧ A → ⟦ tv ⟧ A → C Δ ty tv
change {ty} {tv ⊕ tw} x (i1 y) = Ci1 (change x y)
change {ty} {tv ⊕ tw} x (i2 y) = Ci2 (change x y)
change {ty} {tv}  x  y = CX (delta {ty} {tv} x y)
change-list : {ty tv : U} → ⟦ ty ⟧ A → ⟦ tv ⟧ A → List (C Δ ty tv)
change-list x = return ∘ change x
```

Now, we could S-map change over s:

$$
\begin{aligned}
\mathsf{S\text{-}map\ change}\ s &= \mathsf{Si2}\ (\mathsf{S}\otimes\ (\mathsf{SX}\ (\mathsf{Ci2}\ (\mathsf{CX}\ (\mathsf{i1}\ (4\ ,\ 10), 10))))\ \mathsf{Scp}) \\
&= s'
\end{aligned}
$$

But we are still left with that i1 on the left[1]. Well, this is easy if we could only flip the arguments to C:

```
Sym : UUSet → UUSet
Sym P ty tv = P tv ty
```

And now, we can C-map change with its arguments flipped over the previous s':

```
C-map (flip change) s' = Si2 (S⊗ (SX (Ci2 (CX (Ci1 ((4, 10), 10))) Scp)
```

And bingo! We now have the largest common prefix and information about the differing coproducts on the leaves of this prefix. The whole thing also becomes of a much more expressive type:

```
C-map (flip change) (S-map change s) : S (C (Sym (C (Sym Δ))))
```

We can read the type as: a common prefix from both terms followed by injections into the target term followed by pattern matching on the source term followed by pointwise changes from source to dest. Note the innermost Sym is used to return the type indexes to the correct order. This will make life easier once we start handling fixpoints.

Just like the values of S, we can also define the "apply" relation induced by the values of C and Sym. They are trivial, however. C induces composition with injections, Sym induces converses (which is the relational way of flipping things around). Note that functors are closed with respect to composition, hence, S (C (Sym (C (Sym X)))) makes a functor. Let's call this functor PrePatch X.

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[2].

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 left or right product to be equal to a given value or we can join two alignments together. This is a big source of inneficiency!

Computing alignments is very expensive! In the case we actually have products everywhere, we have a lot of options (hence the list monad!):

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

[1] Previously, we had a *flip* constructor inside our datatype that would let we flip arguments anytime, at will. This is far from ideal, however. In fact, it was this internal *flip* that was causing the first algorithm to not terminate for fixpoints.

[2] 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.

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

```
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))
```

Following our previous example, we could PrePatch-map our alignment function on s', in order to find all alignments of $(4, 10)$ and 10. In this case, this is very easy[3].

```
PrePatch-map align s' =[ Si2 (S⊗ (SX (Ci2 (CX (Ci1 (Ap1° 10 (AX (4,10))))) Scp)  (P1)
                       , Si2 (S⊗ (SX (Ci2 (CX (Ci1 (Ap2° 4 (AX (10,10))))) Scp) ]
                                                                                  (P2)
```

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, I'd like to make a parenthesis here.

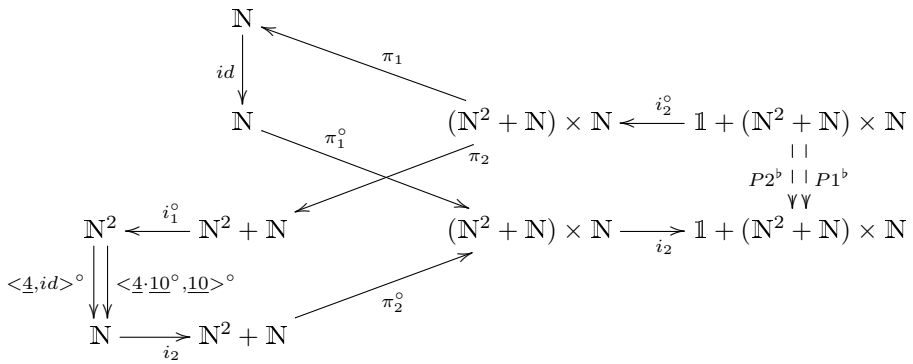
3 Patches as Relations

On Section 2.2 we mentioned that the copy constructor was problematic. Another motivation for removing it and handling everything externally is that we would like to say that the second patch above copies the 10, instead of saying that 10 changes into 10. We can postpone this by changing the relation semantics of Δ. We can say that: $(x, x)^\flat = id$ 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:

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

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

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!

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

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 appliable to much more elements.

This suggests an interesing justification for the cost function. For some reason, looks like we won the lotery 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.

4 Conclusion