Diffing Mutually Recursive Types

A code tour

Victor Cacciari Miraldo

University of Utrecht

December 8, 2016

1 Our Universe

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

$$
\begin{array}{l}
\mathsf{data}\ \mathsf{Atom}\ (n : \mathbb{N}) : \mathsf{Set}\ \mathsf{where} \\
\quad \mathsf{I} : \mathsf{Fin}\ n \quad\ \ \to \mathsf{Atom}\ n \\
\quad \mathsf{K} : \mathsf{Fin}\ ks\# \ \to \mathsf{Atom}\ n
\end{array}
$$

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.

$$
\begin{array}{l}
\pi : \mathbb{N} \to \mathsf{Set} \\
\pi = \mathsf{List} \circ \mathsf{Atom} \\
\\
\sigma\pi : \mathbb{N} \to \mathsf{Set} \\
\sigma\pi = \mathsf{List} \circ \pi
\end{array}
$$

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

$$
\begin{array}{l}
\mathsf{Parms} : \mathbb{N} \to \mathsf{Set}_1 \\
\mathsf{Parms}\ n = \mathsf{Fin}\ n \to \mathsf{Set} \\
\\
[\![\_]\!]_a : \{n : \mathbb{N}\} \to \mathsf{Atom}\ n \to \mathsf{Parms}\ n \to \mathsf{Set} \\
[\![\ \mathsf{I}\ x\ ]\!]_a \qquad A = A\ x \\
[\![\ \mathsf{K}\ x\ ]\!]_a \qquad A = \mathsf{lookup}\ x\ ks \\
\\
[\![\_]\!]_p : \{n : \mathbb{N}\} \to \pi\ n \to \mathsf{Parms}\ n \to \mathsf{Set} \\
[\![\ []\ ]\!]_p \qquad A = \mathsf{Unit} \\
[\![\ a :: as\ ]\!]_p \ \ A = [\![\ a\ ]\!]_a\ A \times [\![\ as\ ]\!]_p\ A \\
\\
[\![\_]\!] : \{n : \mathbb{N}\} \to \sigma\pi\ n \to \mathsf{Parms}\ n \to \mathsf{Set} \\
[\![\ []\ ]\!] \qquad A = \bot \\
[\![\ p :: ps\ ]\!] \quad A = [\![\ p\ ]\!]_p\ A \uplus [\![\ ps\ ]\!]\ A
\end{array}
$$

Note that here, $\mathsf{Parms}\ n$ 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!

$$
\begin{array}{l}
\mathsf{Fam} : \mathbb{N} \to \mathsf{Set} \\
\mathsf{Fam}\ n = \mathsf{Vec}\ (\sigma\pi\ n)\ n \\
\\
\mathsf{data}\ \mathsf{Fix}\ \{n : \mathbb{N}\}(F : \mathsf{Fam}\ n) : \mathsf{Fin}\ n \to \mathsf{Set}\ \mathsf{where} \\
\quad \langle\_\rangle : \forall\{k\} \to [\![\ \mathsf{lookup}\ k\ F\ ]\!]\ (\mathsf{Fix}\ F) \to \mathsf{Fix}\ F\ k
\end{array}
$$

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 $n \to$ 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 : 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
```

$$
\begin{aligned}
&[\![\_]\!]_a : \mathsf{Atom} \to \mathsf{Set}\\
&[\![\ a\ ]\!]_a = \mathsf{interp}_a\ a\ A
\end{aligned}
$$

$$
\begin{aligned}
&[\![\_]\!]_p : \Pi \to \mathsf{Set}\\
&[\![\ p\ ]\!]_p = \mathsf{interp}_p\ p\ A
\end{aligned}
$$

$$
\begin{aligned}
&[\![\_]\!] : \mathsf{U} \to \mathsf{Set}\\
&[\![\ u\ ]\!] = \mathsf{interp}_s\ u\ A
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{UUSet} : \mathsf{Set}_1\\
&\mathsf{UUSet} = \mathsf{U} \to \mathsf{U} \to \mathsf{Set}
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{AASet} : \mathsf{Set}_1\\
&\mathsf{AASet} = \mathsf{Atom} \to \mathsf{Atom} \to \mathsf{Set}
\end{aligned}
$$

$$
\begin{aligned}
&\Pi\Pi\mathsf{Set} : \mathsf{Set}_1\\
&\Pi\Pi\mathsf{Set} = \Pi \to \Pi \to \mathsf{Set}
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{contr} : \forall \{a\ b\}\{A : \mathsf{Set}\ a\}\{B : \mathsf{Set}\ b\}\\
&\quad \to (A \to A \to B) \to A \to B\\
&\mathsf{contr}\ p\ x = p\ x\ x
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{UU}{\to}\mathsf{AA} : \mathsf{UUSet} \to \mathsf{AASet}\\
&\mathsf{UU}{\to}\mathsf{AA}\ P\ a\ a' = P\ (\alpha\ a)\ (\alpha\ a')
\end{aligned}
$$

$$
\begin{aligned}
&{\to}\alpha : \{a : \mathsf{Atom}\} \to [\![\ a\ ]\!]_a \to [\![\ \alpha\ a\ ]\!]\\
&{\to}\alpha\ k = \mathsf{i1}\ (k\ ,\ \mathsf{unit})
\end{aligned}
$$

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 (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:

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

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, (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:

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

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 **??**

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

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.

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 $\alpha : \mathsf{Atom}\ n \to \sigma\pi\ n$; 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 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 → ListI (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 = spine x y, denoted $s^\flat$ are defined by:

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

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

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

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 $⟦\text{2-3-TREE-F}⟧_{\mathsf{Unit}}$ 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.

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:

```
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:

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

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

2.4 Aligning Everything

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].

[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.