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.

```
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 $n \to$ Set, denoted Parms n, which can be seen as a valuation for each type variable.

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

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

TODO

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

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

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

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

TODO

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

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

```
  U : Set
  U = σπ parms#

  Atom : Set
  Atom = Atom' parms#

  Π : Set
  Π = π parms#

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

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

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

⟦_⟧ₚ : Π → Set
⟦ p ⟧ₚ = interpₚ p A

⟦_⟧ : U → Set
⟦ u ⟧ = interpₛ u A

UUSet : Set₁
UUSet = U → U → Set

AASet : Set₁
AASet = Atom → Atom → Set

ΠΠΠSet : Set₁
ΠΠΠSet = 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 $apply_p\ x \equiv \mathsf{just}\ y$.

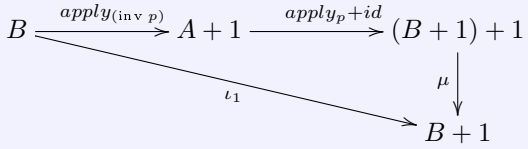
Well, $apply_p$ can be seen as a functional relation (R is functional iff $img\ R \subseteq id$) from A to B. We call this the "application" relation of the patch, and we will denote it by $p^\flat \subseteq A \times B$.

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:



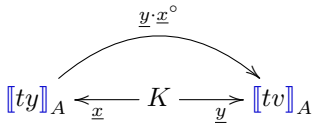
iii) This is hard to play ball with. We want to say, in a way, that $x\ (p^\flat)\ y$ iff $y\ ((\text{inv}\ p)^\flat)\ x$. That is, $(\text{inv}\ p)$ is the actual inverse of p. Using relations, one could then say that $(\text{inv}\ p)^\flat$ is the converse of $(p^\flat)$. That is: $(\text{inv}\ p)^\flat \equiv (p^\flat)^\circ$. 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:



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

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 $s = \mathsf{spine\text{-}cp}\ x\ y$, 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.