

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 Un (n : ℕ) : Set where
  I      : Fin n      → Un n
  K      : Fin ks#    → Un n
  u1     :             Un n
  _ ⊕ _  : (ty tv : Un n) → Un n
  _ ⊗ _  : (ty tv : Un n) → Un 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} → Un 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 (Un 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 $\text{Fin } n \rightarrow \text{Set}$, denoted $\text{Parms } n$, which can be seen as a valuation for each type variable.

In the following sections, we will be dealing with values of $\llbracket ty \rrbracket_A$ 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 : Params n) : Set where
  constructor wb-params
  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. Its module is declared as follows:

```
module RegDiff.Diff.Trivial.Base
  {ks# : ℕ} (ks : Vec Set ks#) (keqs : Vec1 Eq ks)
  {parms# : ℕ} (A : Params 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 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 types, $x : A$ and $y : B$ elements of such types. A *patch* between x and y can be seen as its “application” (partial) function. That is, a relation $e \subseteq A \times B$ such that $\text{img } e \subseteq \text{id}$ (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 accomplished by the Diagonal functor just fine.

```
Δ : UUSet
Δ ty tv =  $\llbracket ty \rrbracket A \times \llbracket tv \rrbracket A$ 
```

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

$$\begin{array}{ccc}
& \xrightarrow{y \cdot \underline{x}^\circ} & \\
\llbracket ty \rrbracket_A & \xleftarrow{\underline{x}} K \xrightarrow{\underline{y}} & \llbracket tv \rrbracket_A
\end{array}$$

Where, for any $A, B \in \text{Set}$ 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 .

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  $\stackrel{?}{=}$  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^b are:

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

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

- Non-cannonicity: $\text{Scp}^b \equiv (\text{S} \otimes \text{Scp } \text{Scp})^b$ 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 $(\text{S } \Delta)$ we would have $(\text{S } (\text{Maybe } \Delta))$, 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 $s = \text{spine-cp } x \ y$ 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 $\mathbb{1} + (\mathbb{N}^2 + \mathbb{N}) \times \mathbb{N}$:

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

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 → List (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 = return (CX (delta {ty} {tv} x y))

```

Now, we could `S-map change` over s :

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

But we are still left with that `i1` on the left¹. 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 (10, (4, 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 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.

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 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 injection. 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`.

In fact, splitting 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.

3 Conclusion

¹Previously, 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.