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:

$$
\begin{array}{l}
\mathsf{data}\ \mathsf{U}_n\ (n : \mathbb{N}) : \mathsf{Set}\ \mathsf{where} \\
\quad \begin{array}{lll}
\mathsf{I} & : \mathsf{Fin}\ n & \rightarrow \mathsf{U}_n\ n \\
\mathsf{K} & : \mathsf{Fin}\ ks\# & \rightarrow \mathsf{U}_n\ n \\
\mathsf{u1} & : & \ \mathsf{U}_n\ n \\
\_\oplus\_ & : (ty\ tv : \mathsf{U}_n\ n) & \rightarrow \mathsf{U}_n\ n \\
\_\otimes\_ & : (ty\ tv : \mathsf{U}_n\ n) & \rightarrow \mathsf{U}_n\ n
\end{array}
\end{array}
$$

Constructor I refers to the n-th type variable whereas K refers to a constant type. Value $ks\#$ is passed as a module parameter. The denotation is defined as:

$$
\begin{array}{l}
\mathsf{Parms} : \mathbb{N} \rightarrow \mathsf{Set}_1 \\
\mathsf{Parms}\ n = \mathsf{Fin}\ n \rightarrow \mathsf{Set}
\end{array}
$$

$$
\begin{array}{l}
[\![\_]\!] : \{n : \mathbb{N}\} \rightarrow \mathsf{U}_n\ n \rightarrow \mathsf{Parms}\ n \rightarrow \mathsf{Set} \\
[\![\ \mathsf{I}\ x\ ]\!]\ A = A\ x \\
[\![\ \mathsf{K}\ x\ ]\!]\ A = \mathsf{lookup}\ x\ ks \\
[\![\ \mathsf{u1}\ ]\!]\ A = \mathsf{Unit} \\
[\![\ ty \oplus tv\ ]\!]\ A = [\![\ ty\ ]\!]\ A \uplus [\![\ tv\ ]\!]\ A \\
[\![\ ty \otimes tv\ ]\!]\ A = [\![\ ty\ ]\!]\ A \times [\![\ tv\ ]\!]\ A
\end{array}
$$

A mutually recursive family can be easily encoded in this setting. All we need is n types that refer to n type-variables each!

$$
\begin{array}{l}
\mathsf{Fam} : \mathbb{N} \rightarrow \mathsf{Set} \\
\mathsf{Fam}\ n = \mathsf{Vec}\ (\mathsf{U}_n\ n)\ n
\end{array}
$$

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

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

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 $\mathsf{Fin}\ n \rightarrow \mathsf{Set}$, denoted $\mathsf{Parms}\ n$, 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 : VecI 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:

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

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 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. We call that a spine:

```
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 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.

The code below is inside the List monad for no good reason. This could be made deterministic and then used with a return clause where needed.

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

  spine : {ty : U} → ⟦ ty ⟧ A → ⟦ ty ⟧ A → List (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  = return (SX (delta {ty} {ty} x y))
```

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

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

3 Conclusion