Diffing Mutually Recursive Types

A code tour

Victor Cacciari Miraldo

University of Utrecht

November 16, 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

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 $n \rightarrow$ 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)
```

In fact, the following sections discuss functionality that is completely independent from the aforementioned parameters. 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 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 Diffing

Intuitively, a *Patch* is some description of a transformation, that can be *applied*, performing the described transformation and can be *computed*, by detecting how to transform one value into another.

That is, the general template we are looking for is:

$Patch \ : \ U \ \rightarrow \ Set$

$diff \ \ : \ \{ty \ : \ U\} \ (x \ y \ : \ [\![ty]\!]) \ \rightarrow \ Patch \ ty$

$apply \ : \ \{ty \ : \ U\} \ \rightarrow \ Patch \ ty \ \rightarrow \ [\![ty]\!] \ \rightarrow \ Maybe \ [\![ty]\!]$

The easiest way to do so is using the diagonal functor:

```
Δ : UUSet
Δ ty tv = ⟦ ty ⟧ A × ⟦ tv ⟧ A
```

3 Conclusion