Meta-programación tipada de meta-programas tipados en Coq

Ignacio Tiraboschi

3 de diciembre de 2019

1. Telescopios

En Mtac2 llamamos *telescopios* a una estructura de datos inductiva que permite expresar una cantidad arbitraria de tipos.

```
Inductive MTele : Type :=
| mBase : MTele
| mTele {X : Type} (F : X -> MTele) : MTele.
```

El tipo `MTele` crea una cadena de abstracciones que toma valores de tipos específicos. Los telescopios, junto con las funciones que lo acompañan será la base para nuestro trabajo.

Eeste tipo puede pensarse en varias jerarquías. La primera sería el telescopio mismo. Un ejemplo puede ser:

```
Let m := @mTele nat (fun _ : nat => mBase)
```

Este telescopio lleva un único tipo, `nat`. Luego existen una infinita cantidad de tipos que dependen de `m`, para continuar el ejemplo elegimos uno.

```
Let A : MTele_Sort SProp m := fun x => x = x.
```

Finalmente, nos interesa un valor de ese tipo, es decir, una prueba.

```
Let a : MTele_val (MTele_C SProp SProp M A) := fun x => ret (eq_refl).
```

2. Aspectos técnicos

Asumiendo que se ha hablado de Coq, es necesario hablar de Mtac2, y mencionar las diferentes estructuras que utilizamos (`MTele_in`, etc).

Con todo eso en la bolsa, el verdadero desafio es el de explicar la forma en que nos aproximamos a esta solución.

Me gustaría hablar de como decidimos la heurística que estabamos utilizando, onda, cómo llegamos a que eso tenía sentido y en que casos se aplicaba. En el mejor de los casos estaría bueno poder idear un estilo de *syntax sugar* para los telescopios. Tal vez sea una buena idea volver a las bases de los telescopios y repensar esto en papel para ver como lo pasamos una fórmula.

Se puede mencionar que tuve que leer código de Mtac2 y tengo unos pull request **mínimos**.

2.1. MFA

Para seguir trabajando, debemos poder representar los tipos monádicos que nos interesan. Para esto definiremos `MFA`.

```
Definition MFA {m} (T : MTele_Ty m) := (MTele_val (MTele_C SType SProp M T)).
```

Dado un telescopio `m`, con n tipos anidados y un tipo `T` de `m`, `MFA T` representa `forall t1 ... tn, M (T t1 ... tn)`

2.2. Bind

```
bind : forall A B : Type, M A -> (A -> M B) -> M B
mbind : forall m : MTele, A B : MTele_Ty m, MFA A -> (A -> MFA B) -> MFA B
```

Figura 1: Signaturas varias

Para comenzar a estudiar el problema es mejor centrarse en casos más simples que podamos razonar. La primera función interesante que podemos *liftear* es `bind` 1.

Es necesario poder entender cual es nuestro objetivo y decidir exactamente qué buscamos modificar de la función. No existe una forma correcta de pensar el tipo, sólo la que nos sirva. En nuestro caso, la idea más simple podemos verla en 1.

En este caso, `(A -> MFA B)` es una función que podemos pensar tiene un tipo equivalente a `(A -> forall t1 ... tn, M (B t1 ... tn))`. Otra forma de pensarlo es con `MTele_In` y `accessor`. De esta forma, podemos expresar `forall t1 ... tn, (A -> M (B t1 ... tn))`. En la figura 2 se puede observar el último caso.

Ahora es momento de definir nuestra nueva función. El primer punto importante es que en el caso de que `m` sea vacío, `mbind` se debe comportar justo como `bind`. El verdadero desafío está en la recursión. Dada la naturaleza recursiva de los telescopios, cada paso recursivo se trata de pelar un tipo, como una cebolla.

```
Fixpoint mbind {m : MTele} : forall {A B : MTele_Ty m},
  MFA A ->
    (MTele_val
      (MTele_In SType (fun a => let A' := a.(acc_sort) A in
                                let B' := a.(acc_sort) B in
                                (A' -> M B')
    ))) -> MFA B :=
  match m with
  | mBase =>
    fun A B ma f => @bind A B ma f
  | @mTele X F =>
      fun A B ma f x => @mbind (F x) (A x) (B x) (ma x) (f x)
  end.
```

Figura 2: El programa `mbind`

Lo importante de esta definición es que funciona para cualquier telescopio y cualquier tipo `A` y `B`. Claramente, nos interesa que `B` efectivamente sea dependiente de todos los argumentos de `m`, mientras tanto, para `A` no es necesariamente importarte, dado que el valor de retorno no lo menciona.

2.3. La Heurística

En el caso de `mbind`, nosotros decidimos cual sería el tipo y adaptamos el código en función de este. Pero nuestro objetivo va más allá. Queremos que cualquier función sea automáticamente *lifteada*. Esto nos obliga a diseñar al programa a través de analisis de tipos, según yo una **heurística** que determine el tipo final.

Podriamos tratar de definir `lift` primero o podríamos tratar de definir el resultado de cada tipo.

A continuación haremos un análisis de como reemplazar cada parte del tipo. Es importante notar que el orden en que esto se define puede cambiar el resultado final, se debe leer este listado asumiendo que se matchea de manera secuencial. Esto se debe a que varios tipos resultan más generales que otros. A nosotros nos interesa poder dividirlos de esta forma por conveniencia.

1. `tyTree_base X`: este es uno de los casos bases, no debemos realizar cambio alguno.

2. `tyTree_M X`: este caso solo va a ser utilizado con el tipo de retorno del programa. Se divide en dos casos: el primero refleja `M X` donde `X` fue reemplazado por `lift`, en el otro caso `X` no fue reemplazado y por lo tanto lo llamaremos el caso constante.

    - `tyTree_M (apply_sort A U)`:
    - `tyTree_M X`: retornamos un `tyTree_MFA T` donde `T` es el tipo telescopico constante.

3. `tyTree_imp X Y`: este es el caso más complicado porque requiere utilizar `lift_in`. Maneja implicaciones sin dependecia entre tipos, mientras existe un caso más general para las dependencias.

4. `tyTree_FA X F` donde `X : Type`, `X` puede seguir siendo el tipo original o haber sido reemplazado por un caso anterior, dado que el tipo se introduce antes que los valores de este mismo. En ambos casos operamos de la misma forma, retornando el tipo `tyTree_FATele1 m F'` donde `F'` es el lifteo de `F`.

5. `tyTree_FAType F`

    - Si `A` se encuentra bajo la mónada, es decir, hay una mención de `A` que se encuentra dentro de `M`, lo reemplazaremos por un `A : MTele_Ty m` para algún `m : MTele` cualquiera.
    - Si no, `A` seguira siendo un `Type`.

Otro aspecto de esto es que este tipo de análisis por casos solo puede realizarse en Mtac2 gracias a `mmatch`, el `match` monádico. Esto es debido a la capacidad de analizar sintacticamente a los tipos.

Para empezar, notamos que los tipos `A` y `B` eran tipos primitivos de Coq, y se convirtieron en `MTele_Ty`, pero ambos tipos aparecían bajo la mónada en la función.