

Meta-programación tipada de meta-programas tipados en Coq

Ignacio Tiraboschi

9 de noviembre 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.

Este 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 desafío 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 *lift*ear 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, $(A \rightarrow \text{MFA } B)$ es una función que podemos pensar tiene un tipo equivalente a $(A \rightarrow \text{forall } t1 \dots tn, M (B \ t1 \dots tn))$. 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 una **heurística** que determine el tipo final.

A continuación analizaremos todos los casos que nos interesan.

- `forall A : Type:`

- Si A se encuentra bajo la mónada en algún punto lo reemplazaremos por **forall** $A : \text{MTele_ty } m$ con un $m : \text{MTele}$ cualquiera.
- Si no, no haremos nada al tipo A .
- **forall** $x : X$ donde $x : \text{Type}$: X puede seguir siendo el tipo original o haber sido reemplazado por la primera regla, dado que el tipo se introduce antes que los valores de este mismo. En ambos casos operamos de la misma forma y simplemente seguimos lifteando.
- $X \rightarrow Y$: este es el caso más complicado porque requiere utilizar MTele_In . Maneja implicaciones sin dependencia entre tipos, mientras existe un caso más general para las dependencias.
-

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.