Meta-programación Tipada de Meta-programas Tipados en Coq

Ignacie Tiraboschi

14 de enero de 2020

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 ℕ (λ _ : ℕ ⇒ mBase)
```

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

```
Let A : MTele_Sort SProp m := λx ⇒ x = x.
```

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

```
Let a : MTele_val (MTele_C SProp SProp M A) := λx ⇒ ret (eq_refl).
```

2. Motivación

Mtac2 nos permite definir funciones monádicas. Estas cuentan con ciertas ventajas. Un ejemplo de un meta-programa es el siguiente.

```
Definition list_max_ℕ :=
  mfix f (l: list ℕ ) : l <> nil → M ℕ :=
    mtmmatch l as l' return l' <> nil → M ℕ with
    | [? e] [e] =m> λnonE ⇒ M.ret e
    | [? e1 e2 l'] (e1 :: e2 :: l') =m> λnonE ⇒
      let m := Nat.max e1 e2 in
      f (m :: l') cons_not_nil
    | [? l' r'] l' ++r' ⇒ (* cualquier cosa *)
    end.
```

Esta función calcula el máximo de una lista de números ℕ : `Set`. Dado que en el último caso del `match` monádico analiza una expresión con una función, y no un constructor, es imposible implementar esto sin Mtac2.

Ahora supongamos que deseamos parametrizar ℕ y tener una función que acepte múltiples conjuntos. Sea

```
Definition max (S: Set) : M (S → S → S) :=
  mmatch S in Set as S' return M (S' → S' → S') with
  | ℕ ⇒ M.ret Nat.max
  end.
```

la función que retorna la relación máximo en un conjunto S. A primera vista nuestra idea podría fucionar.

```
Definition list_max (S: Set) :=
  max ← max S; (* error! *)
  mfix f (l: list S) : l' <> nil → M S :=
    mtmmatch l as l' return l' <> nil → M S with
    | [? e] [e] =m> λnonE⇒ M.ret e
    | [? e1 e2 l'] (e1 :: e2 :: l') =m> λnonE ⇒
      m ← max e1 e2;
      f (m :: l') cons_not_nil
    end.
```

Al intentar que Coq interprete la función veremos que lstlisting no lo acepta. Esto es debido a la signatura de `bind`. Nuestro `mfix` no puede unificarse a `M B`, ya que tiene tipo `f : forall (l: list S) l' <> nil → M S` .

```
bind : forall A B, M A → (A → M B) → M B.
```

Solucionar esta situación específica no es un problema. Una alternativa es introducir los parámetros de la función y beta-expandir la definición del fixpoint. Otra es codificar un nuevo bind que tenga el tipo necesario. El problema será que este solo soluciona el problema actual, y ante cualquier variación necesitaremos volver a solucionarlo.

Es por eso que nuestro proyecto es la codificación de un nuevo meta-programa `lift` que automaticamente puede generalizar meta-programas con las dependencias necesarias para que sea utilizado en cualquier contexto.

3. Lift

Denominamos `lift` a la función desarrollada en este trabajo. La misma tiene la tarea de agregar dependencias a meta-programas de manera quasi automática: solo requiere un telescopio.

Lift se basa en analizar los tipos de las funciones y modificarlos a voluntad, generando así nuevos meta-programas que añaden los tipos del telescopio como dependencias. El signficado real de esto va ser trabajado en este capítulo a través de ejemplos que mostrarán el comportamiento de lift.

3.1. Lifteando funciones simples

La función monádica más simple es `ret : ∀ A, A → M A`, uno de los operadores monádicos. Supongamos que nos interesa tener

```
ret^ : ∀ (A : ℕ → ℕ → Type), (∀ n n', A n n') → (∀ n n', M (A n n'))
```

Ahora, nuestro trabajo es poder definir un telescopio que se amolde a la información que necesitamos agregar. En este caso el telescopio es el siguiente: `t := [n : ℕ ;> n' : ℕ ]` dado que queremos que A dependa de dos ℕ. Efectivamente `ret^ := lift ret t`.

Este ejemplo es simple porque no involucra funciones. Pero en ese caso, estudiemos que sucede con bind.

```
bind : ∀ A B, M A → (A → M B) → M B
```

Bind es el operador monádico más importante y es central en estos problemas. Supongamos que queremos utilizamos el telescopio `t := [T : Type ;> l : list T]` donde `list T` es el tipo de las listas con elementos de T. Al momento de liftearlo, no es obvio cual debería ser el resultado, así que veamoslo.

```
bind^ : ∀ A B : ∀ T : Type, list T → Type,
        (∀ (T : Type) (l : list a), M (A T l)) →
        (∀ (T : Type) (l : list a), A T l → M (B T l)) →
        (∀ (T : Type) (l : list a), M (B T l))
```

Lo interesante es que si miramos la definición de esta función es sumamente simple.

```
λ (A B : ∀ T : Type, list T → Type)
  (ma : ∀ (T : Type) (l : list T), M (A T l))
```

```
(f : ∀ (T : Type) (l : list a), A T l → M (B T l))
(T : Type) (l : list T) ⇒
bind (A T l) (B T l) (ma T l) (f T l)
```

Esta definición es efectivamente la que buscabamos y funciona perfectamente. El otro aspecto que seguiremos observando es que Lift no genera información necesaria en la función destino, lo que es bueno en cuanto a rendimiento.

4. Aspectos Técnicos

En esta sección discutiremos los aspectos técnicos de Lift. Comenzaremos discutiendo su funcionamiento básico y de ahí escalaremos a los detalles más profundos.

4.1. TyTree

En terminos generales Lift es fixpoint sobre un telescopio con un gran análisis por casos sobre los tipos. Entonces surge un problema: ¿Cómo podemos hacer pattern matching en los tipos de manera sintáctica? La solución es utilizar un reflejo de los mismos, de manera de que podamos expresarlos de manera inductiva.

```
Inductive TyTree : Type :=
| tyTree_val {m : MTele} (T : MTele_Ty m) : TyTree
| tyTree_M (T : Type) : TyTree
| tyTree_MFA {m : MTele} (T : MTele_Ty m) : TyTree
| tyTree_In (s : Sort) {m : MTele} (F : accessor m → s) : TyTree
| tyTree_imp (T : TyTree) (R : TyTree) : TyTree
| tyTree_FATele {m : MTele} (T : MTele_Ty m)
  (F : ∀ t : MTele_val T, TyTree) : TyTree
| tyTree_FATele1 (m : MTele) (F : ∀ (T : MTele_Ty m), TyTree) : TyTree
| tyTree_FAValue (T : Type) (F : T → TyTree) : TyTree
| tyTree_FAType (F : Type → TyTree) : TyTree
| tyTree_base (T : Type) : TyTree
.
```

Con este tipo podemos reescribir todas las signaturas de funciones. Varios de los constructores, como por ejemplo `tyTree_MFA` o `tyTree_FATele`, tendrán sentido más adelante, dado que reflejan elementos en funciones lifteadas.

Una de las propiedades principales de este reflejo es que a primera vista parece que un tipo puede escribirse de múltiples formas en `TyTree`, pero los tratamos de manera distinta y por eso podemos plantear una biyección entre `Type` y `TyTree`.

Ahora tomaremos una función `f` e iremos modificando si signatura para mostrar esta reflección de tipos. Para simplificar escribiremos `P` $\equiv$ `Q` para expresar que un tipo es equivalente a un `TyTree`.

`f :` $\mathbb{N} \to \mathbb{N} \to \mathbb{N}$

Su tipo traducido es

`f :` imp_{tT} $(\mathrm{base}_{tT}\ \mathbb{N})$ $(\mathrm{imp}_{tT}\ (\mathrm{base}_{tT}\ \mathbb{N})\ (\mathrm{base}_{tT}\ \mathbb{N}))$

Dado que no hay dependencias de tipos, podemos utilizar imp_{tT} . Ahora si parametrizamos $\mathbb{N}$ por cualquier tipo.

$\forall$ `A, A → A → A`
$\equiv$
FAType_{tT} $(\lambda$ `A` $\Rightarrow$ imp_{tT} $(\mathrm{base}_{tT}$ `A`$)$ $(\mathrm{imp}_{tT}$ $(\mathrm{base}_{tT}$ `A`$)$ $(\mathrm{base}_{tT}$ `A`$)))$

Con FAType_{tT} podemos observar claramente la dependencia de `A`.

Para expresar la dependencia de un valor utilizamos $\mathrm{FAValue}_{tT}$.

$\forall$ `A (B : A → Type) (a : A), (B a)`
$\equiv$
FAType_{tT} $(\lambda$ `A` $\Rightarrow$ FAType_{tT} $(\lambda$ `B : A → Type` $\Rightarrow$ $\mathrm{FAValue}_{tT}$ `A` $(\lambda$ `a` $\Rightarrow$ base_{tT} `(B a)`$))))$

4.2. 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)`

4.3. 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 (λ a ⇒ let A' := a.(acc_sort) A in
                              let B' := a.(acc_sort) B in
                              (A' → M B')
    ))) → MFA B :=
  match m with
  | mBase ⇒
    λ A B ma f ⇒ ·bind A B ma f
  | ·mTele X F ⇒
      λ 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.

4.4. 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_FAValue 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.