Meta-programación Tipada de Meta-programas Tipados en Coq

Ignacie Tiraboschi

16 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 := λ 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) := λ x ⇒ 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 λ nonE ⇒ M.ret e
    | [? e1 e2 l'] (e1 :: e2 :: l') ⇒m λ nonE ⇒
      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 de `mtmmatch`, `match` monádico generalizado, analiza una expresión con una función, y no un constructor, es imposible implementar esto sin Mtac2. Notar que tampoco podemos utilizar `mmatch` ya que el tipo que retornamos es `l' <> nil → M ℕ`.

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, es decir, conceptualmente no es incorrecta.

```
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 λ nonE⇒ M.ret e
    | [? e1 e2 l'] (e1 :: e2 :: l') ⇒m λ nonE ⇒
      m ← max e1 e2;
      f (m :: l') cons_not_nil
    end.
```

Al intentar que Coq interprete la función veremos que esta función no tipa. 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 ambas soluciones son específicas al problema, entonces en cada situación debemos volver a implementar alguno de estos recursos.

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 el contexto. En nuestro ejemplo, con Lift podemos generalizar `bind` consiguiendo un nuevo meta-programa que se comporta como la función original pero con una signatura distinta, permitiendo su uso.

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 añadiendo dependencias triviales en los tipos que se encuentran bajo la mónada, generando así nuevos meta-programas más generales. El signficado real de este funcionamiento será trabajado en este capítulo a través de ejemplos que mostrarán el comportamiento de lift.

3.1. Lifteando funciones simples

Una de las funciones monádica más simples 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 ]`$_t$ 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]`$_t$ donde `list T` es el tipo de las listas con elementos de tipo 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 T), M (A T l)) →
```

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

Lo más imporante es que si observamos 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 innecesaria en la función destino.

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 términos generales Lift es un 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_FATeleVal {m : MTele} (T : MTele_Ty m)
  (F : ∀ t : MTele_val T, TyTree) : TyTree
| tyTree_FATeleType (m : MTele) (F : ∀ (T : MTele_Ty m), TyTree) : TyTree
| tyTree_FAVal (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_FATeleVal`, 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`.

Utilizaremos $\texttt{Const}_t$ para expresar `tyTree_Const`, donde `Const` representa alguno de los nombres de los constructores de la definición de `TyTree`.

Ahora tomaremos una función `f` e iremos modificando si signatura para mostrar este reflejo de tipos. Para simplificar escribiremos $\texttt{P} \equiv \texttt{Q}$ para expresar que un tipo es equivalente a un `TyTree` aunque no sea técnicamente correcto en Coq.

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

Su tipo traducido es

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

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

$\forall\ \texttt{A},\ \texttt{A} \to \texttt{A} \to \texttt{A}$
$\equiv$
$\mathrm{FAType}_{tT}\ (\lambda\ \texttt{A} \Rightarrow \mathrm{imp}_{tT}\ (\mathrm{base}_{tT}\ \texttt{A})\ (\mathrm{imp}_{tT}\ (\mathrm{base}_{tT}\ \texttt{A})\ (\mathrm{base}_{tT}\ \texttt{A}))$

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

Para expresar la dependencia de un valor utilizamos FAVal_{tT}.

$\forall\ \texttt{A}\ (\texttt{B} : \texttt{A} \to \texttt{Type})\ (\texttt{a} : \texttt{A}),\ (\texttt{B a})$
$\equiv$
$\mathrm{FAType}_{tT}\ (\lambda\ \texttt{A} \Rightarrow \mathrm{FAType}_{tT}\ (\lambda\ \texttt{B} : \texttt{A} \to \texttt{Type} \Rightarrow \mathrm{FAVal}_{tT}\ \texttt{A}\ (\lambda\ \texttt{a} \Rightarrow \mathrm{base}_{tT}\ (\texttt{B a}))))$

El centro de nuestro trabajo son las funciones monádicas, esto significa utilizar M_{tT}.

$\texttt{ret} : \forall\ \texttt{A},\ \texttt{A} \to \texttt{M A}$
$\equiv$
$\texttt{ret} : \mathrm{FAType}_{tT}\ (\lambda\ \texttt{A} \Rightarrow \mathrm{imp}_{tT}\ (\mathrm{base}_{tT}\ \texttt{A})\ (\mathrm{M}_{tT}\ \texttt{A}))$

Es importante notar que podemos encontrar usos de M_{tT} en múltiples secciones de la signatura, solo se matcheará M_{tT} en Lift con el retorno de la función.

4.2. El tipo de LIFT

Analicemos la definición de LIFT.

```
Fixpoint lift (m : MTele) (U : ArgsOf m) (p l : bool) (T : TyTree) :
  forall (f : to_ty T), M m:{ T : TyTree & to_ty T} := ...
```

Dentro de esta signatura vemos elementos conocidos como el telescopio `m : MTele` que anuncia las dependencias, un `T : TyTree` que representa el tipo de la función a *liftear* y la `f : to_ty T` de tipo `T`. También conocemos

```
M m:{ T : TyTree & to_ty T}
(* Σ-type con un tipo y un elemento del mismo *)
```

En el retorno conseguimos un nuevo `T'` : `TyTree` que representa la signatura de la función `f` lifteada, y un elemento de tipo `T'`, es decir, la nueva función.

Ahora, ¿que representan los argumentos que no mencionamos? Es **fácil** comprender quienes son `p` y `l`.

- `p`: la polaridad. Comienza con valor `true`. Este solo se modifica cuando matcheamos imp_{tT} en LIFT. No es útil para `lift_in` y representa...
  - `p` = `true`:
- `l`: comienza en `false`. Representa si nos encontramos a la derecha o izquierda de una implicación de manera inmediata. Es útil para...

En cuanto a `U : ArgsOf m`, `U` representa los argumentos que el telescopio añade, y es nuestro truco para poder hacer funcionar LIFT, ya que nos permite transportar argumentos de manera descurrificada. Lo que sucede es que, cuando encontremos un tipo `A` cualquiera en nuestra función, ese tipo puede o no estar bajo la mónada. En el caso de estarlo está en nuestro interés modificarlo, es decir, que deje de ser un `Type`, siendo dependiente de los argumentos de nuestro telescopio. Por eso, con `U` y otras funciones de telescopios podemos conseguir este comportamiento.

4.3. TyTrees monádicos

Ahora que ya vimos los comportamientos de LIFT debemos centrarnos en cómo representamos con `TyTree` las funciones lifteadas. Esto significa entender aún más detalles de los tipos que dependen de telescopios.

En este caso observamos el tipo de `ret` lifteado. Es importante notar que está parametrizado a cualquier telescopio, lo que significa que liftear una función no necesariamente necesita de un telescopio específico.

```
λ m : MTele ⇒
  FATeleType_tT m
    (λ A : MTele_Ty m ⇒
      imp_tT (In_tT Type_s (λ a : accessor m ⇒ acc_sort a A))
      (MFA_tT A))
```

En este podemos observa el uso de FATeleType_{tT} , In_{tT} y MFA_{tT}

Con FATeleType_{tT} podemos introducir tipos telescopios, se comporta de igual manera que FAType_{tT} .

`MTele_In` nos permite adentrarnos a un tipo telescópico, momentaneamente introduciendo todos los argumentos con un `accessor` y trabajando sobre el tipo de manera directa. Dado que no tenemos interés real es usar estos argumentos del telescopio, no tenemos que hacer demasiado trabajo, simplemente generamos tipos de manera trivial, es decir, ignorando los argumentos.

Utilizamos MFA_{tT} para representar tipos monádicos cuantificados. Definimos `MFA` en Mtac2 de la siguiente manera.

```
Local Definition MFA {t} (T : MTele_Ty t) := (MTele_val
  (MTele_C Type_sort Prop_sort M T)).
```

Sea `t` un telescopio cualquiera de largo n, un `MFA T` representará

```
∀ x_1 ... x_n, M (T x_1 ... x_n)
```

Finalmente, en el caso anterior, interpretando los tipos de una manera más matemática, tomamos un valor $\forall\ x_1 \ldots x_n,\ A\ x_1 \ldots x_n$ y retornamos $\forall\ x_1 \ldots x_n,\ M\ (T\ x_1 \ldots x_n)$. Si concretizamos el telescopio conseguiremos una signatura más parecida a la matemática y la función resultante será muy simple. Para mostrar esto supongamos que tenemos un telescopio $t := [x_1 : T_1;> x_2 : T_2;> x_3 : T_3]_t$. Luego,

```
ret^ :=
λ (A : T_1 → T_2 → T_3 → Type)
  (x : ∀ (t_1 : T_1) (t_2 : T_2) (t_3 : T_3), A t_1 t_2 t_3)
  (x_1 : T_1) (x_2 : T_2) (x_3 : T_3) ⇒ r (A x_1 x_2 x_3) (A x_1 x_2 x_3)
: ∀ T : T_1 → T_2 → T_3 → Type,
  (∀ (t_1 : T_1) (t_2 : T_2) (t_3 : T_3), T t_1 t_2 t_3) →
  ∀ (t_1 : T_1) (t_2 : T_2) (t_3 : T_3), M (T t_1 t_2 t_3)
```

Solo nos falta hablar de FATeleVal_{tT} . Tomemos una función de ejemplo con la siguiente signatura.

```
∀ (T : Type) (R : T → Type) (t : T), M (R t)
≡
FAType_tT (λ T ⇒ FAType_tT (λ R : T → Type ⇒ FAVal_tT T (λ t ⇒ M_tT (R t))))
```

La traducción será muy directa.

FAType_{tT} (λ `T` $\Rightarrow$ FATeleType_{tT} (λ `R` : `T` $\rightarrow$ `Type` $\Rightarrow$ FATeleVal_{tT} `T` (λ `t` $\Rightarrow$ M_{tT} (`R t`))))

Notar que el primer constructor no es reemplazado ya que `T` no se encuentra bajo la mónada en ningún momento.