Meta-programación Tipada de Meta-programas Tipados en Coq

Ignacio Tiraboschi

21 de enero de 2020

Índice general

Capítulo 1

Una introducción

1.1. Introducción

1.2. Coq

En este capitulo introduciremos las características más relevantes del asistente de pruebas Coq. El objetivo de este capitulo es introducir todos los conceptos que utilizaremos más adelante, pero esto significa que no es una introducción completa.

1.2.1. La historia de Coq

El desarrollo de Coq comenzó en 1984 con el apoyo de INRIA como el trabajo de Gérard Huet y Thierry Coquand. En ese momento Coquand estaba implementado un lenguaje llamado *Calculus of Constructions* cuando el 1991 una nueva implementación basada en un Calculus of Inductive Constructions extendido comenzó a ser desarrollado tomando el nombre de Coq.

Ahora mismo Coq es desarrollado por más de 40 desarrolladores activos y es reconocido como unos de los asistentes de prueba principales.

1.2.2. Los lenguajes Coq

Coq no es técnicamente un lenguaje de programación, si no un asistente de pruebas. Pero podemos encontrar múltiples lenguajes dentro de Coq que nos permiten expresarnos.

- *Gallina*: este es el lenguaje de especificación de Coq. Permite desarrollar teorías matemáticas y probar especificaciones de programas.

Utilizaremos extensivamente este lenguaje para definir nuestros programas en Mtac2.

- *Ltac*: este el lenguaje en que se definen las *tácticas* de Coq. Dado que Coq está centrado en las tácticas, Ltac es una de las partes centrales del aparato.
- *Vernacular*.

Aunque Coq no es un lenguaje de programación propiamente dicho, este puede ser utilizado como un lenguaje de programación funcional. Estos programas serán especificados en Gallina. Dada la naturaleza de Coq como provador de teoremas, estos programas son funciones puras, es decir, no producen efectos secundarios y siempre terminan.

1.2.3. La teoría de Coq

Calculus of Inductive Constructions es la base de Coq. Este es un cálculo lambda tipado de alto orden y puede ser interpretado como una extensión de la correspondencia Curry-Howard.

Llamaremos *Terms* (o términos) a los elementos básicos de esta teoría. Terms incluye *Type*, *Prop*, variables, tuplas, funciones, entre otros. Estás son algúnas de las herramientas que utilizaremos para escribir nuestros programas.

1.2.4. Set vs. Prop vs. Type

1.2.5. Tipos de datos y Funciones

Ahora aprenderemos a codificar nuestros programas funcionales en Coq. Lo primero que debemos entender es que operamos sobre *términos* y algo es un término si tiene tipo. Coq provee muchos tipos predefinidos, por ejemplo `unit`, `bool`, `nat`, `list`, entre otros. A continuación estudiaremos cómo definir tipos y funciones.

Veamos cómo se define el tipo `bool`:

```
Inductive bool : Set :=
  | true : bool
  | false : bool.
```

Como podemos observar, es un tipo inductivo, especificado por la keyword `Inductive`, con dos constructores `true` y `false`. De por sí, el tipo `bool` no posee un significado hasta que nosotros lo proveamos de uno. Podemos ahora intentar definir algunos operadores booleanos.

```
Definition andb (b1 b2:bool) : bool := if b1 then b2 else false.
Definition orb (b1 b2:bool) : bool := if b1 then true else b2.
Definition implb (b1 b2:bool) : bool := if b1 then b2 else true.
Definition negb (b:bool) := if b then false else true.
```

Las definiciones de funciones no recursivas comienzan con el keyword `Definition`. La primera se llama `andb` y toma dos booleanos como argumentos y retorna un booleano. Se utiliza la notación `if b then x else y` para matchear sobre los booleanos de manera más fácil. Finalmente podemos definir una función más interesante.

```
Definition Is_true (b:bool) :=
  match b with
    | true ⇒ True
    | false ⇒ False
  end.
```

Ahora, veamos un tipo con un ingrediente un poco más complicado, `nat`.

```
Inductive nat : Set :=
  | O : nat
  | S : nat → nat.
```

Notemos que el constructor `S` es una función que recibe un término de tipo nat, es decir, `nat` es un tipo recursivo. Por ejemplo el término `S (S O)` es de tipo `nat` y representa al número 2.

Para continuar con este desarrollo, veamos el tipo de `list` que es polimórfico.

```
Inductive list (A : Type) : Type :=
  | nil : list A
  | cons : A → list A → list .
```

Este tipo es un tipo polimórfico dado que requiere de un `A : Type`. Por ejemplo, una posible lista es `cons (S O) nil : list nat` que representa a la lista con un único elemento 1 de tipo `nat`.

Definiremos una función que añade un elemento a una lista.

```
Definition add_list {A} (x : A) (l : list A) : list A :=
  cons x l.
```

Dado que el tipo `A` puede ser inferido facilmente por Coq, utilizamos llaves a su alrededor para expresar que sea un argumento implícito. En el cuerpo de la función solo utilizamos `cons`, uno de los constructores de `list`, para añadir un elemento delante de `l`.

Ahora nos interesa definir la función `length` que retorna el largo de una lista.

```
Fixpoint len {A} (l : list A) : nat :=
match l with
| []  ⇒ O
| x ::  xs ⇒ S (len xs)
end.
```

Coq está diseñado de forma que necesitamos utilizar el keyword `Fixpoint` para poder definir funciones recursiva. Aquí Coq está encontrando el argumento decreciente de la función `len` y por eso acepta nuestra definición. El cuerpo de `len` inspecciona a `l` y lo *matchea* con el caso correspondiente. Utilizamos `S` y `O`, los constructores de `nat` para expresar el valor de retorno.

1.2.6. Tipos dependientes

Una de las herramientas más importantes que hay en Coq son los tipos dependientes. Estos nos permiten hablar de elementos que dependen de otros anteriores. Por ejemplo, puede ser de nuestro interés hablar de números positivos, en otras palabras, `x : nat` tal que `x <> O`. En este caso, `x <> O` es una prueba que depende de `x` y solo existirá cuando `x` sea mayor a 0.

Para hablar de un ejemplo práctico de tipos dependientes, hemos elegido la función `head` que retorna la cabeza de una lista. Comencemos con la versión más simple, donde utilizamos un valor default `d` para el caso en que la lista es vacía.

```
Definition head_d {A} (l : list A) (d : A) : A :=
  match l with
  | []  ⇒ d
  | x ::  xs ⇒ x
  end.
```

El problema de esta solución es que a excepción de que `d` sea un valor único, no hay manera de saber si la función retornó realmente la cabeza de la lista.

La segunda opción es utilizar el tipo `option`.

```
Inductive option A : Type :=
| None : option A
| Some : A → option A.
```

Con este tipo auxiliar podemos reescribir `hd`.

```
Definition head_o {A} (l : list A) : option A :=
  match l with
  | []  ⇒ None
  | x ::  xs ⇒ Some x
  end.
```

Esta solución es mejor que la anterior pero sigue sufriendo de una deficiencia. Dado que `head_o` retorna un `option` no sabemos si el resultado de esta función será realmente un elemento o si será el constructor vacío, por lo que todas las funciones que utilicen a `head_o` deben también utilizar `option`.

Esto nos lleva a nuestra última solución. Esta requiere que nos aseguremos que la lista `l` no es vacia, es decir, `l <> []`. Pero para entenderla debemos ver dos cosas más: Σ-*types* y `Program`. Intuitivamente, los Σ-*types* son tuplas donde el argumento de la derecha es dependiente del de la izquierda. A continuación, la definición.

```
Inductive sig (A : Type) (P : A → Prop) : Type :=
  exist : ∀ x : A, P x → {x : A | P x}
```

Se utiliza la notación `{x : A | P x}` para expresar `sig A (λ x ⇒ P)`.

`Program` es una libreria que permite progamar en Coq como si fuera un lenguaje de programación funcional mientras que se utiliza una especificación tan rica como se desee y probando que el codigo cumple la especificación utilizando todo el mecanizmo de Coq. En nuestro caso utilizaremos `Program` para codificar `head` de una manera casi transparente.

```
Program Definition head {A}
(l : list A | [] <> l ) : A :=
  match l with
  | [] ⇒ !
  | x :: xs ⇒ x
  end.
```

Como podemos observar, la única diferencia es que la signatura de `head` especificamos que `l` es una lista junto con una prueba que muestra que no es vacía.

Este trabajo se centra en modificar meta-programas añadiendo tipos dependientes donde sea necesario, por eso sumamente importante ejercitar este concepto.

1.3. Mtac2

1.3.1. Mónadas

1.3.2. Confección de Meta-programas

1.3.3. Meta-tácticas

1.3.4. Alternativas

1.4. Telescopios

En Mtac2 llamamos *telescopios* a una estructura de datos inductiva que permite expresar una secuencia de tipos posiblemente dependientes de largo arbitrario.

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

El tipo `MTele` crea una cadena de abstracciones. Este se codifica a través de funciones, lo que permite que sean dependientes, es decir, un telescopio puede tener elementos que dependan de elementos anteriores.

Los telescopios, junto con las funciones que lo acompañan, serán claves a la hora de poder expresar nuestro problema y nuestra solución.

Los telescopios no se caracterizan por su simpleza. Para comenzar a estudiarlos podemos pensarlos en jerarquías. La primera jerarquía serían los telescopios en sí, elementos de tipo `MTele`. Definiremos la siguiente notación para poder referirnos a estos de manera más simple.

```
mBase ≡ [ ]_t
mTele (λ T : Type ⇒ mTele R : T → Type) ≡ [T : Type;> R : (T → Type)]_t
mTele (λ T : Type ⇒ mTele t : T) ≡ [T : Type;> t : T]_t
```

Las otras jerarquías entran en juego cuando utilizamos las distintas funciones telescopicas.

La principal función es `MTele_Sort`. Dado un telescopio `t` de largo n y un `s : Sort`, esta función computa $\forall$ x_1 ... x_n, `s`, es decir, retorna el tipo expresado por ese telescopio y el `Sort` dado. Utilizaremos `MTele_Ty` y `MTele_Prop` para expresar `MTele_Sort` $\texttt{Type}_s$ y `MTele_Sort` $\texttt{Prop}_s$ respectivamente. Elementos de este tipo serán de la segunda jerarquía.

También debemos poder referirnos a valores de estos tipos telescópicos. Esto signfica que dado un elemento de tipo `MTele_Sort s t`, podamos conseguir un elemento de tipo `s`. En terminos matemáticos, λ x_1 ... x_n ⇒ `T` x_1 ... x_n

con `T : MTele_Sort s t`. Consideraremos que estos elementos pertenecen a la tercera y última jerarquía.

Deseriamos que este fuera el final de los telescopios, pero nos falta el último ingrediente que nos proveerá del poder que buscamos. Con `MTele_In` podemos ganar acceso a múltiples tipos y valores telescopios al mismo tiempo, así siendo capaces de computar un nuevo tipo telescópico.

Capítulo 2

Lift

2.1. Motivación

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

```
Definition list_max_nat :=
  mfix f (l: list nat) : l <> nil → M nat :=
    mtmmatch l as l' return l' <> nil → M nat 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 `nat : 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 nat`.

Ahora supongamos que deseamos parametrizar $\mathbb{N}$ 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
  | nat ⇒ 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.

2.2. 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.

2.2.1. 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 : nat → nat → 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 : nat ;> n' : nat`$]_t$ dado que queremos que A dependa de dos $\mathbb{N}$. 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.

2.3. 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.

2.3.1. 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 :=
| val{m : MTele} (T : MTele_Ty m) : TyTree
| M (T : Type) : TyTree
| MFA{m : MTele} (T : MTele_Ty m) : TyTree
| In(s : Sort) {m : MTele} (F : accessor m → s) : TyTree
| imp(T : TyTree) (R : TyTree) : TyTree
| FATeleVal{m : MTele} (T : MTele_Ty m)
  (F : ∀ t : MTele_val T, TyTree) : TyTree
| FATeleType(m : MTele) (F : ∀ (T : MTele_Ty m), TyTree) : TyTree
| FAVal(T : Type) (F : T → TyTree) : TyTree
| FAType(F : Type → TyTree) : TyTree
| base(T : Type) : TyTree
.
```

Con este tipo podemos reescribir todas las signaturas de funciones. Varios de los constructores, como por ejemplo `MFA` o `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`. Utilizamos la función `to_ty : TyTree → Type` para transformar un `TyTree` en su `Type` correspondiente. Notar que esta función no es monádica, y eso es principal, ya que nos permite utilizar la función en las signaturas que definimos. En cambio, la función `to_tree : Type → M TyTree` necesariamente es monádica ya que debemos hacer un analisis sintáctico en los tipos de Coq. Haremos uso extensivo de `to_ty`.

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 `P ≡ Q` para expresar que un tipo es equivalente a un `TyTree` aunque no sea técnicamente correcto en Coq.

```
f : nat → nat → nat
```

Su tipo traducido es

```
f : imp(base nat) (imp (base nat) (base nat))
```

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

```
∀ A, A → A → A
```

```
≡
FAType(λ A ⇒ imp(base A) (imp (base A) (base A))
```

Con `FAType` podemos observar claramente la dependencia de `A`.
Para expresar la dependencia de un valor utilizamos `FAVal`.

```
∀ A (B : A → Type) (a : A), (B a)
≡
FAType(λ A ⇒ FAType(λ B : A → Type ⇒ FAValA (λ a ⇒ base(B a))))
```

El centro de nuestro trabajo son las funciones monádicas, esto significa utilizar `M`.

```
ret : ∀ A, A → M A
≡
ret : FAType(λ A ⇒ imp(base A) (M A))
```

Es importante notar que podemos encontrar usos de `M` en múltiples secciones de la signatura, solo se matcheará `M` en Lift con el retorno de la función.

2.3.2. El tipo de Lift

Analicemos la definición de Lift.

```
Fixpoint lift (m : MTele) (U : ArgsOf m) (p l : bool) (T : TyTree) :
  ∀ (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, ¿qué 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` 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. Esto quiere decir que transporta a los argumentos en un "contenedor". Lo que sucede es que cuando encontremos un tipo `A` cualquiera en nuestra función, este tipo puede o no estar bajo la mónada. En el caso de estarlo debemos modificarlo, es decir, reemplazar `A : Type` por `A : MTele_Ty t` con `t : MTele`.

2.3.3. TyTrees monádicos

Ahora nos centraremos en cómo representar funciones lifteadas con `TyTree`. Esto significa entender aún más detalles de los tipos dependientes 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 ⇒
  FATeleTypem
    (λ  A : MTele_Ty m ⇒
      imp(In Type_s (λ  a : accessor m ⇒ acc_sort a A))
      (MFA A))
```

Este es el tipo lifteado de `ret`. En este podemos observa el uso de `FATeleType`, `In` y `MFA`.

Con `FATeleType` podemos introducir tipos telescopios, se comporta de igual manera que `FAType`.

`MTele_In` nos permite adentrarnos a un tipo telescópico, momentáneamente 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. Esto es expresado por `acc_sort a A` que en verdad está produciendo el tipo ∀ x_1 ... x_n, `A` x_1 ... x_n.

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

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

Sea `t : MTele` de largo n y `T : MTele_Ty t`, `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 $\forall\ \mathtt{x}_1\ \ldots\ \mathtt{x}_n, \mathtt{A}\ \mathtt{x}_1\ \ldots\ \mathtt{x}_n$ y retornamos $\forall\ \mathtt{x}_1\ \ldots\ \mathtt{x}_n, \mathtt{M}\ (\mathtt{T}\ \mathtt{x}_1\ \ldots\ \mathtt{x}_n)$.

En el caso de `ret`, la signatura $\mathtt{In}\ \mathtt{Type}_s\ (\lambda\ \mathtt{a} : \mathtt{accessor\ m} \Rightarrow \mathtt{acc\_sort\ a\ A})$ es simplemente equivalente a $\mathtt{val}\ \mathtt{A}$ pero Coq no puede inferir esto directamente. La forma en que hemos definido LIFT utiliza esta forma más general en todos los casos.

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 $\mathtt{t} := [\mathtt{x}_1 : \mathtt{T}_1;> \mathtt{x}_2 : \mathtt{T}_2;> \mathtt{x}_3 : \mathtt{T}_3]_t$. Luego,

$$
\begin{aligned}
&\mathtt{ret\hat{}} :=\\
&\lambda\ (\mathtt{A} :\ \mathtt{T}_1 \to \mathtt{T}_2 \to \mathtt{T}_3 \to \mathtt{Type})\\
&\quad (\mathtt{x} :\ \forall\ (\mathtt{t}_1 : \mathtt{T}_1)\ (\mathtt{t}_2 : \mathtt{T}_2)\ (\mathtt{t}_3 : \mathtt{T}_3), \mathtt{A}\ \mathtt{t}_1\ \mathtt{t}_2\ \mathtt{t}_3)\\
&\quad (\mathtt{x}_1 : \mathtt{T}_1)\ (\mathtt{x}_2 : \mathtt{T}_2)\ (\mathtt{x}_3 : \mathtt{T}_3) \Rightarrow \mathtt{r}\ (\mathtt{A}\ \mathtt{x}_1\ \mathtt{x}_2\ \mathtt{x}_3)\ (\mathtt{A}\ \mathtt{x}_1\ \mathtt{x}_2\ \mathtt{x}_3)\\
&: \forall\ \mathtt{T} :\ \mathtt{T}_1 \to \mathtt{T}_2 \to \mathtt{T}_3 \to \mathtt{Type},\\
&\quad (\forall\ \ (\mathtt{t}_1 : \mathtt{T}_1)\ (\mathtt{t}_2 : \mathtt{T}_2)\ (\mathtt{t}_3 : \mathtt{T}_3), \mathtt{T}\ \mathtt{t}_1\ \mathtt{t}_2\ \mathtt{t}_3) \to\\
&\quad \forall\ (\mathtt{t}_1 : \mathtt{T}_1)\ (\mathtt{t}_2 : \mathtt{T}_2)\ (\mathtt{t}_3 : \mathtt{T}_3), \mathtt{M}\ (\mathtt{T}\ \mathtt{t}_1\ \mathtt{t}_2\ \mathtt{t}_3)
\end{aligned}
$$

Solo nos falta hablar de `FATeleVal`. Tomemos una función de ejemplo con la siguiente signatura.

$$
\begin{aligned}
&\forall\ (\mathtt{T} :\ \mathtt{Type})\ (\mathtt{R} :\ \mathtt{T} \to \mathtt{Type})\ (\mathtt{t} : \mathtt{T}),\ \mathtt{M}\ (\mathtt{R}\ \mathtt{t})\\
&\equiv\\
&\mathtt{FAType}\ (\lambda\ \ \mathtt{T} \Rightarrow \mathtt{FAType}\ (\lambda\ \ \mathtt{R} : \mathtt{T} \to \mathtt{Type} \Rightarrow \mathtt{FAVal}\ \mathtt{T}\ (\lambda\ \ \mathtt{t} \Rightarrow \mathtt{M}\ (\mathtt{R}\ \mathtt{t}))))
\end{aligned}
$$

La traducción será muy directa.

$$\mathtt{FAType}\ (\lambda\ \ \mathtt{T} \Rightarrow \mathtt{FATeleType}\ (\lambda\ \ \mathtt{R} : \mathtt{T} \to \mathtt{Type} \Rightarrow \mathtt{FATeleVal}\ \mathtt{T}\ (\lambda\ \ \mathtt{t} \Rightarrow \mathtt{M}\ (\mathtt{R}\ \mathtt{t}))))$$

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

2.3.4. El algoritmo

A continuación haremos un recorrido paso a paso de como `ret` es lifteado.

1. Matcheamos $\mathtt{FAType}\ \mathtt{F}$ donde $\mathtt{F} :\ \mathtt{Type} \to \mathtt{TyTree}$. Generamos un tipo arbritario `A` y calculamos $\mathtt{is\_m}\ (\mathtt{F}\ \mathtt{A})\ \mathtt{A}$. Esta función retornará `true` ya que efectivamente el tipo `A` se encuentra bajo la mónada en la signatura $\mathtt{F}\ \mathtt{A}$. Entonces generamos un nuevo $\mathtt{A} :\ \mathtt{MTele\_Ty\ t}$ aplicamos LIFTde manera recursiva sobre $\mathtt{F}\ (\mathtt{apply\_sort\ A\ U})$.

2. Ahora debemos liftear algo del siguiente tipo.

```
F (apply_sort A U) ≡
imp(base (apply_sort A U)) (M (apply_sort A U)))
```

Nuestra expresión matcheará con el caso `imp` en el cual comenzaremos evaluando `contains_u m U X` donde `X` será `base(apply_sort A U)`. En nuestro caso es claro que se cumple dado que hemos reemplazado `A` por un tipo descurrificado. Estos nos lleva a tener que utilizar `lift_in`.

3. Tendremos una llamada `lift_in U (base (apply_sort A U))` matcheando el caso correspondiente. `lift_in` retornará una Σ-Type con un valor `F : accessor t` $\to$ Type_s y una prueba de que el tipo `base (apply_sort A U)` es igual a `F (uncurry_in_acc U)`. `uncurry_in_acc U` nos retorna el accesor trivial de `U`. Lo importante es que sabemos que `F (uncurry_in_acc U)` es igual al lado izquierdo de la implicación de `ret`.

4. Ahora volvemos a LIFT y generamos un valor `x : X'` con

   `X' :=` $\texttt{MTele\_val (MTele\_In Type}_s\texttt{ F)}$

   Es decir, un valor `x` del tipo resultante de liftear `X` en el paso anterior. Lo que resta es tomar nuestra función `f` de tipo `X'` $\to$ `Y` y liftear. Esto signfica liftear `f x`. No entraremos en los detalles específicos de como expresar `f x` ya que son meros métodos de escribirlo y no portan un valor real.

5. El último paso es `lift t U Y (f x)` sabiendo que `Y = M (apply_sort A U)`, entonces matcheamos con el caso correspondiente. Primero, lo que hacemos es abstraer a `U` de `f` obteniendo

   `f =` $(\lambda$ `U` $\Rightarrow$ `to_ty (tyTree_M (apply_sort A U)))`

   y luego aplicamos `f` a `curry`, de esa manera, la función pasa a tener tipo `to_ty (MFAA)`.

   `f :` $\forall$ $\texttt{x}_1$ $\texttt{x}_n$ $\Rightarrow$ $\texttt{MFA(A x}_1$ ... $\texttt{x}_n\texttt{)}$

6. Finalmente, LIFT retorna una `T' : TyTree` y `retˆ : to_ty (T')` con

   `T' = FATeleType(`λ `A` $\Rightarrow$ `imp(val A) (MFA A)).`

Capítulo 3

Conclusión

3.1. Conclusión

Capítulo 4

Apéndice

4.1. Apéndice A

4.2. Bibliografía