UNIVERSIDAD NACIONAL DE CÓRDOBA
Facultad de Matemática, Astronomía, Física y Computación

GENERALIZACIÓN DE META-PROGRAMAS CON TIPADO DEPENDIENTE EN MTAC2

IGNACIO TIRABOSCHI

DIRECTOR:

DR. BETA ZILIANI

LICENCIADO EN CIENCIAS DE LA COMPUTACIÓN

Marzo 2020

RESUMEN

Los meta-lenguajes de programación son una parte esencial de los asistentes de pruebas. Estos meta-lenguajes deben lidiar con multiples problemas y para esto utilizan diferentes mecanismos. En particular, el meta-lenguaje Mtac2 en Coq utiliza mónadas como una forma de añadir tipado descriptivo a las tácticas de Coq.

Estos programas monádicos no son fáciles de desarrollar. En parte, vincular cómputos monádicos puede ser una tarea complicada con el sistema de tipos de Coq, donde se utilizan tipos dependientes. Solucionar estos problemas requiere de esfuerzo programacional inútil, no relacionado al objetivo.

Utilizando Mtac2 podemos crear un nuevo metaprograma que generalice la signatura de las funciones de forma casi automática, permitiendo una comunicación fluida entre declaraciones monádicas.

Con este trabajo, el programador puede concentrarse en lo verdaderamente importante y olvidarse de los detalles fuertemente restrictivos de la naturaleza de Coq, con un esfuerzo pequeño.

ABSTRACT

Meta-languages are an essential part of theorem provers. This meta-languages have to deal with several problems, hence they employ different mechanisms. Particularly speaking, the meta-language Mtac2 on Coq, utilizes monads as a means to add rich typed tactics to Coq.

This monadic programs are hard to develop. Partly, *binding* monadic elements when Coq's type system uses dependent types is not simple. Solving this involves great effort on developing useless code that is not related to the actual problem.

Using Mtac2 we can create a new meta-program that can almost automatically generalize the signature of functions, allowing a seamless communication between monadic statements.

With this work, the programmer can focus on truly important tasks, forgetting the highly restrictable nature of Coq, with little effort.

ÍNDICE GENERAL

ÍNDICE DE FIGURAS

LISTINGS

Parte I

INTRODUCCIÓN

1 INTRODUCCIÓN

El uso de *asistentes de prueba* es cada vez mayor. Múltiples de ellos cuentan con metalenguajes de programación que permiten la automatización del proceso de desarrollo de pruebas. Dentro del espectro de los asistentes de prueba, Coq [**CIC**] es una las opciones más conocidas, siendo Mtac2 [**DBLP:journals/pacmpl/KaiserZKRD18**] uno de los metalenguajes disponibles. En Mtac2, el uso de mónadas nos permite generar lo que se denomina *tácticas*, en este caso, tipadas a diferencia de las usuales. Estas metafunciones monádicas utilizan operadores monádicos para poder reflejar cómputos a través de la concatenación de subcómputos.

El uso de estas funciones monádicas es complejo. El caso en el que nos centraremos es cuando nuestros operadores monádicos no sean lo suficientemente generales para poder *vincular* elementos monádicos. En principio, estos problemas no surgirán necesariamente, pero si el desarrollador está utilizando tipos dependientes, utilizará cuantificadores para poder expresar estas dependencias. La problemática es que los operadores monádicos de Mtac2 no permiten tomar, ni retornar, elementos cuantificados.

Este trabajo podría haberse resumido a codificar vesiones más expresivas de los dos operadores monádicos, `bind` y `ret`. Sin embargo, hemos preferido desarrollar una solución más general. Lo que tenemos es una metafunción que toma a otras metafunciones como argumento y retorna "versiones cuantificadas" de las mismas, de manera casi automática. De esta forma, los dos operadores a los que nos referimos, simplemente necesitan ser pasados como argumentos a nuestra nueva metafunción, que hemos denominado Lift.

1.1 ASISTENTES DE PRUEBA

Históricamente, una prueba matemática es una sucesión de instrucciones, más específicamente, reglas de inferencia, en un lenguaje más formal que el natural y que, al ser aplicadas en orden, partiendo de una serie de hipótesis, pueden llegar a una conclusión, el teorema. Si estas reglas son utilizadas correctamente y las hipótesis no se contradicen, entonces todo resultado al que lleguemos puede ser considerado verdadero. Efectivamente, estos pasos representan una prueba de dicho resultado.

Sin embargo, en las pruebas matemáticas no explicitamos cada regla que utilizamos, pues la lectura de la demostración resultaría muy tediosa y, claramente, no es necesario que sea extremadamente formal

para que el lector pueda comprender la prueba. Pero entonces, estas pruebas no tiene la rigurosidad absoluta que desearíamos, podemos decir que estas siguen siendo, en cierto punto, informales. En este contexto, es claro que los lectores no pueden dar un veredicto exacto sobre el valor de verdad de un teorema.

Tenemos la suerte de que las computadoras son muy adecuadas para este tipo de trabajos rigurosos. De aquí surge la motivación de desarrollar herramientas para verificar estos resultados matemáticos, pero que también puedan asistir al desarrollador en el proceso. Algunas de estas herramientas son Coq [**CIC**], Isabelle [**DBLP:books/sp/NipkowPW02**], Agda [**DBLP:conf/tphol/BoveDN09**], Lean [**DBLP:conf/cade/MouraKADR15**] y HOL4 [**DBLP:conf/tphol/SlindN08**].

1.2 COQ

Coq es uno de los asistentes de prueba más utilizados. Este cuenta con numerosos casos de éxito, tanto en matemática [**DBLP:conf/ascm/Gonthier07**], como en computación [**DBLP:journals/pacmpl/0002JKD18**].

Uno de los aspectos más importantes de Coq es que las pruebas se codifican a través de la concatenación de *tácticas*. Supongamos que tenemos un teorema a probar. Pensaremos este teorema como una meta o un objetivo. A través del uso de tácticas, modificaremos la meta, también llamada *goal*. Las tácticas generarán nuevas metas, que representarán pasos intermedios de la demostración. Por ejemplo, podemos pensar la inducción matemática como una táctica, la cual tomará nuestra meta actual, generando dos nuevas metas: el caso base y el paso inductivo. En este caso, de una meta generamos dos nuevas. El trabajo para desarrollar una prueba consiste en utilizar estas tácticas para reducir nuestra meta hasta que esta sea verdadera.

El aspecto más importante de las tácticas, al menos en nuestro caso, es que el programador puede desarrollar sus propias tácticas con el objetivo de ser asistido a la hora de escribir la prueba. Estás tácticas son, por defecto, definidas en el metalenguaje Ltac [**DBLP:conf/lpar/Delahaye00**].

Existen múltiples tipos de tácticas que intentan diferentes cosas, e inclusive, con diferentes filosofías. Un ejemplo es la táctica CoqHammer [**DBLP:journals/jar/CzajkaK18**], una de las más conocidas en Coq. Esta tiene como objetivo demostrar todo automáticamente. Aunque esta táctica pueda resultar muy poderosa, también conlleva muchos problemas, como mencionaremos más adelante.

1.3 MTAC2

El metalenguaje MTAC2 [**DBLP:journals/pacmpl/KaiserZKRD18**] tiene como objetivo reemplazar a Ltac. En este sentido, es uno de tantos que intentan lo mismo: Ltac2 (Cap. Ltac2 en Manual de Referencia de

Coq 8.11 [**Coq**]), Template-Coq [**DBLP:conf/itp/AnandBCST18**], Rtac [**DBLP:conf/esop/MalechaB16**] y Coq-Elpi [**tassi:hal-01637063**].

Volviendo a CoqHammer, esta táctica presenta varias deficiencias. En parte, esta táctica no es rápida: ejecutarla puede tardar múltiples horas, y aún así fallar. Si todo funciona, no debemos preocuparnos. Si falla, no tenemos certeza de por qué fue. Más aun, podríamos haber utilizado la táctica de manera incorrecta y no saberlo, ya que las tácticas de Ltac no son tipadas. Es decir, podemos intentar utilizar cualquier táctica en cualquier momento, pero solo funcionarán las tácticas adecuadas. Por eso, una de las características más importantes de Mtac2 es que tiene tácticas tipadas. Estas, solo pueden ser utilizadas en el momento en que la meta que queremos probar se aproxima a lo que nuestra táctica acepta. Además nos asegura de qué forma será la meta resultante de la ejecución. Por último, en el caso de fallar, tendremos certeza de cual fue la razón.

1.4 DESARROLLO DEL TRABAJO

En el Capítulo 2, podemos encontrar una introducción a Coq, con todos los conceptos principales que utilizaremos más adelante. Primero introduciremos los diferentes lenguajes que residen en Coq, donde dos de ellos serán de gran importancia: Ltac y Gallina. Veremos como las tácticas de Coq, definidas en Ltac, nos permiten codificar pruebas matemáticas. En cuanto a Gallina, lo utilizaremos para poder definir nuestras funciones y tipos. En esta introducción, cubriremos tipos inductivos, recursivos y dependientes. Utilizaremos Σ-tipos para poder expresar dependencias en los tipos y definir funciones que ofrezcan más certezas al depender de pruebas. Además, discutiremos cuales son las consecuencias de que Coq sea un lenguaje puro.

En el Capítulo 3 encontraremos una introducción a Mtac2. Primero, introduciremos el concepto de mónadas, ya que Mtac2 define una mónada para reflejar efectos secundarios. Hablaremos de la creación de metaprogramas, cubriendo las versiones monádicas de pattern matching y recursión. Luego, discutiremos la falta de garantías que estos metaprogramas proveen en comparación con programas puros de Coq. Finalmente, definiremos *telescopios* en Mtac2. Esta signatura es la que nos permitirá diseñar nuestra metafunción Lift.

En la segunda parte de este trabajo encontraremos tres capítulos:

- En el Capítulo 4 expondremos con mayor precisión el problema que estamos resolviendo con una función modelo. Esta función tiene el problema de que no podrá ser parametrizada automáticamente. Ahí entrará en juego Lift, generando las metafunciones necesarias para que la función modela pueda ser parametrizada.
- En el Capíutlo 5 analizaremos a la metafunción Lift de manera conceptual. Analizaremos las signaturas de las metafunciones

originales, y llegaremos a la signatura deseada, dejando en evidencia nuestra necesidad de generalizarlas. Nos concentraremos en comprender la lógica de la solución.

- En el Capítulo 6 haremos un estudio en profundidad de la función LIFT. Comenzaremos estudiando el tipo auxiliar `TyTree`, el cual nos permitirá describir las signaturas de nuestras funciones con un tipo inductivo. Luego, definiremos múltiples funciones y tipos auxiliares que serán suficientes para poder definir LIFT. Y cerraremos este capítulo con dos ejecuciones de LIFT, paso por paso.
- En el Apéndice A, encontraremos la definición de todas las funciones y tipos utilizados en LIFT, junto con la definición completa de LIFT.

2 COQ

En este capítulo introduciremos las características más relevantes del asistente de prueba interactivo Coq. El objetivo aquí, es introducir todos los conceptos que utilizaremos más adelante.

El desarrollo de Coq comenzó en 1984 como el trabajo de Gérard Huet y Thierry Coquand. En ese momento, Coquand estaba implementado un lenguaje llamado *Calculus of Constructions* [**DBLP:journals/iandc/CoquandH88**]. En 1991, Coquand creó una derivación del trabajo anterior, *Calculus of Inductive Constructions* [**CIC**]. Esta teoría comenzó a ser desarrollada y el producto final es lo que hoy llamamos Coq.

Ahora mismo Coq es desarrollado por más de 40 personas de forma activa y es reconocido como unos de los principales asistentes de prueba. Como un asistente de prueba, la orientación de Coq es la de permitir la escritura totalmente formal de teoremas y pruebas, asegurándonos de su correción. Coq utiliza un pequeño *kernel*, o núcleo, que verifica las pruebas desarrolladas por el humano. De esta forma, el humano no está encargado de verificar la prueba, solo de escribirla.

El ejemplo más conocido de formalización en Coq es el teorema de los cuatro colores [**DBLP:conf/ascm/Gonthier07**] hecho por **DBLP:conf/ascm/Gonthier07** en **DBLP:conf/ascm/Gonthier07**.

2.1 LOS LENGUAJES COQ

Coq no es técnicamente un lenguaje de programación, sino, un asistente de prueba. Podemos encontrar múltiples lenguajes dentro de Coq que nos permiten expresarnos, como los que describiremos brevemente a continuación:

GALLINA Este es el lenguaje de especificación de Coq [**Gallina**]. Permite desarrollar teorías matemáticas y probar especificaciones de programas. Utilizaremos extensivamente un lenguaje muy similar a Gallina para definir nuestros programas en MTAC2.

LTAC El lenguaje en que se definen las *tácticas* de Coq. Es el metalenguaje por defecto de Coq. Introducido por **DBLP:conf/lpar/Delahaye00** en Coq 7.0. Ltac [**DBLP:conf/lpar/Delahaye00**] es central en Coq y hasta algunos lo consideran una de las principales razones del éxito de este. Antes de Ltac, los usuarios de Coq tenían que escribir los *proof terms* a mano, o sino, utilizaban un conjunto de tácticas muy primitivas que ejecutaban reglas muy básicas. Ltac le proporcionó a los usuarios una forma de escribir sus

propias tácticas, al unir las tácticas básicas gracias a un conjunto expresivo de combinadores, permitiendo así el desarrollo de formalizaciones más complejas en Coq.

VERNACULAR Este lenguaje contiene los comandos básicos de Coq con los que enviamos directivas al intérprete. En lo que resta del trabajo utilizaremos varios de estos comandos.

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

2.2 LA TEORÍA DE COQ

El trabajo *Calculus of Inductive Constructions* [**CIC**] es la base de Coq, 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. Estas son algunas de las herramientas que utilizaremos para escribir nuestros programas.

2.3 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 que 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 bool es un tipo inductivo, especificado por el comando `Inductive` de Vernacular, con dos constructores `true` y `false`. De por sí, el tipo `bool` no posee un significado hasta que nosotros lo proveamos de uno. Ahora podemos 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 comando `Definition`. La primera se llama `andb` y toma dos booleanos

como argumentos, retornando un booleano. Se utiliza la notación `(if b then x else y)` para matchear sobre los booleanos de manera sencilla. Finalmente, podemos definir una función más interesante.

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

Listing 2.1: Definición de `is_true`

La definición del Listing 2.1 es muy importante. Nos muestra claramente que `true` y `True` no son lo mismo. Como vimos, `true` representa el valor de verdad booleano. En Coq, las proposiciones son representadas por tipos, y sus pruebas como programas de ese tipo. El tipo `True` es un término de tipo `Prop`, es decir, una proposición y, las proposiciones se consideran verdaderas si hay términos de ese tipo. Cómo su nombre lo indica, `True` representa la proposición siempre verdadera. La función `Is_true` nos permite ver esa diferencia lógica entre proposiciones y expresiones booleanas.

El tipo `False` representa a la proposición que siempre es falsa, es decir, que nunca podremos probar. En caso de probar `False`, sin ninguna hipótesis, estaríamos mostrando que Coq es *inconsistente*.

```
Inductive True : Prop :=
 I : True.

Inductive False : Prop :=.
```

Listing 2.2: Definición de `True` y `False`

Ahora, veamos un tipo inductivo recursivo.

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

Notemos que el constructor `S`, que representa "sucesor", es una función que recibe un término de tipo nat, es decir, `nat` es un tipo *recursivo*, a diferencia de `bool`. 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, aparte de ser recursivo, es polimórfico.

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

Este tipo es un tipo polimórfico dado que requiere de un `A : Type` para ser un tipo. Antes de conocer ese `A`, `list` es una función de `Type` en `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`.

Ahora tratemos de definir una función polimórfica. 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 fácilmente 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 `len` que retorna el largo de una lista.

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

En Coq, utilizamos el comando `Fixpoint` para poder definir funciones recursivas. Para que Coq pueda verificar la correción de las pruebas, es necesario que las funciones que forman parte de la definición de las tácticas utilizadas, sean totales. Para garantizar su totalidad, Coq impone restricciones en la definición de funciones recursivas: las llamadas recursivas deben ser sobre argumentos decrecientes. Aquí, Coq está encontrando el argumento decreciente de la función `len`, verificando la terminación de `len`. El cuerpo de la función inspecciona a `l` y lo *matchea* con el caso correspondiente. Utilizamos `S` y `O`, los constructores de `nat`, para expresar el valor de retorno.

2.4 TIPOS DEPENDIENTES

Una de las herramientas más importantes de 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, más formalmente, `x : nat` tal que `x <> 0`. En este caso, `x <> 0` es una proposición que depende de `x` y, será verdadera solo cuando `x` sea mayor a cero.

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 sea 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`. Este tipo es una mónada y tiene dos constructores, uno que nos permite indicar que tenemos un valor y otro que representa el error. De esta forma, podemos escribir la siguiente función.

```
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 elemento de tipo `option` no sabemos si el resultado de esta función será realmente un elemento o si será el constructor vacío hasta que inspeccionemos al valor, por lo que todas las funciones que utilicen a `head_o` deben también utilizar la mónada `option`.

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

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

Listing 2.3: Definición de Σ-*types*

Se utiliza la notación `{x : A |P x}` para expresar `sig A (fun x ⇒ P)`. Utilizaremos este tipo de manera extensa para nuestras definiciones.

La librería `Program` permite programar en Coq como si fuera un lenguaje de programación funcional mientras que, por detrás, se utiliza una especificación tan fuerte como se desee y probando que el código cumple la especificación de manera automática. 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.
```

En la signatura de `head` podemos ver el uso del Σ-tipo. En el caso de la lista vacía, utilizamos `!` para expresar que este caso es inalcanzable, permitiendo que `Program` genere una obligación. En esta versión, podemos claramente ver la utilidad de los tipos dependientes. Solo podemos utilizar esta función en caso de que la lista sea no vacía.

2.5 TÁCTICAS

En esta sección, hablaremos de pruebas, metas (*goals*) y tácticas. Para esto, introduciremos un ejemplo que nos facilite entender estos conceptos.

```
Lemma sub_0_r : forall n, n - 0 = n.
Proof. intro n. case n; [ | intro n']; reflexivity. Qed.
```

Listing 2.4: Teorema y prueba en Coq

En el Listing 2.4, definimos un lema enunciando que restar cero a cualquier número n da como resultado n. La resta está definida por recursión en el primer argumento y, al ser `n` una variable, no es posible reducir la expresión `(n - 0)`; por lo tanto, la prueba de este teorema no es inmediata. Sería distinto en el caso de que el primer argumento fuera el número o y el segundo `n`: o-n; esta prueba sería trivial dada la definición de la resta. Ya que la resta está definida por pattern matching en el primer argumento, esta igualdad no es automáticamente cierta computando. Por eso, debemos hacer análisis por casos.

El código comienza con el comando `Lemma`, donde efectivamente especificamos lo que queremos probar. Luego, utilizamos el comando `Proof` para indicar el inicio de la prueba, la cual será resuelta a través de la concatenación de tácticas de Ltac. Estas tácticas transforman el *proof-state* incrementalmente construyendo un *proof-term*, la prueba. Podemos pensar al proof-state como el estado parcial de la prueba. Aplicando tácticas podemos modificar este estado, y al mismo tiempo ir creando nuestra prueba, generalmente llamada proof-term.

Después de `Proof`, Coq genera una `goal`, una meta. Internamente, una meta en Coq es representada con una *metavariable*. Esta metavariable tiene un tipo, en concreto el lema que queremos probar y será representada con $?g$. En este caso, nuestra meta $?g$ tiene tipo $\forall$ `n, n - 0 = n`.

Para introducir la variable n, utilizamos `intro`. Esta instancia a $?g$ como `fun n:nat` $\Rightarrow ?g_1$ donde el tipo de $?g_1$ es `n - 0 = n`.

Una vez introducida la variable, el próximo paso es hacer análisis por casos en ella. Con `case` podemos analizar a n según los constructores de su tipo. Para el primer caso: `0 - 0 = 0` es trivial. El segundo caso es $\forall$ `n':nat, S n' - 0 = S n'`, para el cual, primero introduciremos n', y luego, por la naturaleza inductiva de la resta, será igualmente trivial para Coq. La táctica `case` retorna estas dos sub-metas, las cuales componemos, con el operador de composición (el punto y coma), con las tácticas listadas en `[ | intro n']`. La primer sub-meta es resuelta por la táctica a la izquierda del `|`, que es implicitamente la táctica identidad (`idtac`), mientras que la segunda sub-meta es resuelta por la táctica a

la derecha de `|`, `intro n'`. La salida de esta composición son de nuevo dos tácticas las que de nuevo compondremos con `reflexivity`. Esto significa que aplicaremos `reflexivity` a ambas tácticas resultantes, resolviéndolas trivialmente por computación.

2.6 INTERFAZ INTERACTIVA

Cuando se dice que Coq es un asistente *interactivo* nos referimos a que Coq nos puede ayudar a desarrollar la prueba en cierta medida. Por ejemplo, supongamos que queremos probar el siguiente teorema. Notemos que el comando `Theorem` se utiliza para definir teoremas.

```
Theorem le_S (n : nat) : n <= S n.
```

Listing 2.5: Teorema ejemplo

Para escribir la prueba del teorema, utilizaremos dos lemas ya definidos en Coq.

```
le_0_n : forall n : nat, 0 <= n
le_n_S : forall n m : nat, n <= m → S n <= S m
```

Listing 2.6: Teoremas `le_0_n` y `le_n_S`

Al entrar a alguno de los editores de textos compatibles con Coq (Emacs, Visual Studio Code o CoqIDE), cargamos el teorema y Coq entrará al modo interactivo, en el cual nos mostrará el estado actual de la prueba. En este caso, comenzamos con la hipótesis `n : nat` ya en nuestro contexto y una única meta $?g$ de tipo `n <= S n`. Ahora aplicamos inducción en `n` obteniendo dos submetas: $?g_1$ con tipo `0 <= S 0` y $?g_2$ con tipo `S n <= S (S n)`. Para la primera meta $?g_1$, utilizamos `apply` para aplicar el teorema `le_0_n` instanciado con `S n`, con lo que se soluciona automáticamente la submeta.

La segunda sub-meta se resuelve de una manera similar, utilizando el teorema `le_n_S` y la hipótesis inductiva que llamaremos `IHn`.

En la Figura 2.1, podemos observar como se nos presenta esta información.

Coq cuenta con muchos comandos que se usan constantemente. Estos comandos son parte del "lenguaje" introducido anteriormente: Vernacular. Algunos de los comandos más importantes son los siguientes.

CHECK Con `Check` podemos consultar el tipo de un término. Cuando es llamado en modo prueba, el término es chequeado en el contexto local de la submeta.

Comando	*Respuesta*
`Check S 0.`	`S 0 : nat`
`Check nat.`	`Set`

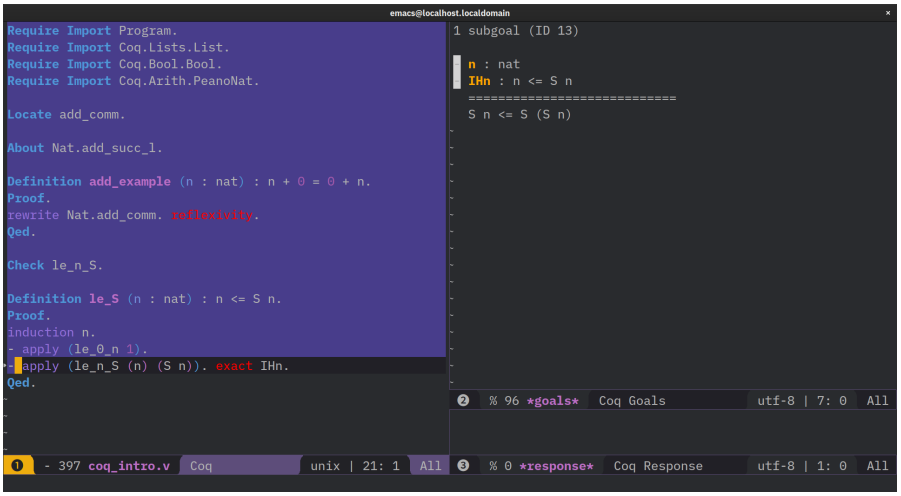

Figura 2.1: Ejemplo de interfaz de Coq

ABOUT El comando `About` provee información general. Por ejemplo, podemos consultar la signatura de una función, la de un tipo, o consultar el tipo de un constructor. A diferencia de `Check`, no podemos evaluar cualquier termino. Por ejemplo, `About S 0.` retornará un error.

Comando	*Respuesta*
`About S.`	`S : nat ->nat`
`About list.`	`list : Type ->Type`

PRINT Podemos pensar a `Print` como un `About` que además nos muestra la definición completa. De esta manera, podemos inspeccionar elementos ya definidos.

Comando	*Respuesta*
`Print list.`	`Inductive list (A : Type) : Type := ...`

LOCATE Con este comando podemos encontrar donde han sido definidos los elementos.

Comando	*Respuesta*
`Locate bool.`	`Inductive Coq.Init.Datatypes.boo`

EVAL Con `Eval` podemos evaluar términos. La manera en que los evaluemos o reduzcamos es elegida por nosotros.

Comando	*Respuesta*
`Eval compute in 1 + 1.`	`= 2 : nat`
`Eval cbv in (fun x =>x * (x + 1)) 2.`	`= 6 : nat`

En este caso, `compute` se traduce a β-reducir las funciones y `cbv` se refiere a "call-by-value". En este contexto, `compute` y `cbv` reducen a lo mismo. En otros casos será muy importante diferenciar entre estos, incluyendo `cbn` ("call-by-name").

Podemos encontrar una lista completa de estos comandos en Vernacular Commands.

3

MTAC2

MTAC2 [**DBLP:journals/pacmpl/KaiserZKRD18**] es un metalenguaje de programación para Coq. Esto quiere decir que complementa a Coq, permitiendo "usar Coq" de una manera distinta. En el trabajo, nos centramos en ampliar MTAC2 y, por eso, es importante que veamos que nos permite hacer y cómo.

3.1 MÓNADAS

Las mónadas son uno los métodos que utilizan los lenguajes de programación funcional para la representación de efectos secundarios. MTAC2 define una mónada `M` que es la que permite añadir una serie de características muy útiles a la hora de desarrollar metaprogramas. Las tácticas presentes en MTAC2 son metaprogramas monádicos.

Con la mónada `M : Type → Prop` podemos referirnos a la versión monádica de un tipo cualquiera. A partir de los tipos monádicos, pasamos a tener elementos monádicos. Estos elementos reflejan pasos computacionales y se construyen a través de dos funciones: `bind`compone pasos computacionales y `return` o `ret` los envuelve en la mónada. Por ejemplo, podemos tener el tipo `M nat` que expresa posibles valores de números naturales. La expresión `ret 5` expresa un valor de `M nat`.

3.2 CONFECCIÓN DE METAPROGRAMAS

Las tácticas de MTAC2 son metaprogramas. Estos programas se caracterizan por ser monádicos, es decir, utilizan mónadas para reflejar efectos secundarios. Esto se amplía a muchas características útiles, pero se debe pagar un costo. A continuación, intentaremos comprender las limitaciones impuestas por el metalenguaje.

Comenzaremos analizando `mmatch`. Como ya vimos en Gallina, `match` es puro, o sea, cuando lo utilizamos, necesitamos matchear todos los casos del constructor y a su vez no podemos matchear en términos que no sean constructores del tipo. Mientras tanto, `mmatch` nos permite matchear más libremente. Esto significa que podemos matchear en expresiones sintácticas de manera de separarlas muy específicamente para nuestra conveniencia. Un ejemplo puede ser el siguiente.

```
Definition test_match (n : nat) : M nat :=
 mmatch n with
 | [? x y] add x y ⇒ ret n
 | 0 ⇒ ret (S 0)
 | [? n'] S n' ⇒ ret (S (S n'))
```

```
 end.
```

En el primer caso del `mmatch` del programa `test_match`, vemos que no estamos matcheando uno de los constructores de `nat`. Esta es la gran diferencia con el `match` de Coq. En el `match` original de Coq, solo podemos matchear a los constructores del tipo del argumento. Mientras tanto, acá estamos analizando `add x y`, que tiene tipo `nat`, pero no es uno de los constructores del tipo.

El único detalle extraño que podemos encontrar es que en dos de los casos tenemos unos corchetes antes de la expresión. Esto se utiliza para decirle a MTAC2 que estas variables están siendo introducidas al contexto.

Para hacer programas recursivos utilizaremos `mfix`. Existen múltiples variantes para una cantidad distinta de argumentos recursivos: `mfix1`, `mfix2`, etcetera.

Un ejemplo puede ser el de `map`.

```
Definition map {A} {B} (t : A → B) : ∀ (l : list A), M (list B)
    :=
mfix1 m (l : list A) : M list B :=
 mmatch l with
 | nil ⇒ ret nil
 | [? x xs] x::xs ⇒ xs' ← m xs;
                    ret ((t x)::xs')
 end.
```

En `map` solo tenemos un argumento recursivo, que será la lista que estamos mapeando. En el ejemplo, también vemos el uso de la notación $\leftarrow$. Esta indica que estamos *bindeando* a la variable `xs'` con el cómputo `m xs`, es decir, utilizando la función `bind` para conectar estos cómputos. En otros ejemplos es posible que veamos otra notación: `;;`. Está notación también indica el uso de `bind`, con la diferencia de que no nos interesa el valor que estamos vinculando. Usualmente, lo observaremos al utilizar la función `print`, ya que esta función retorna un argumento de tipo `M unit`.

Los últimos dos elementos que necesitamos entender son `mtmmatch` y `mfix`. Estas son versiones de `mmatch` y $\texttt{mfix}_n$, respectivamente.

En primer lugar, utilizaremos `mtmmatch` para realizar pattern matching monádico, con la diferencia de que los valores de retorno pueden ser dependientes, es decir, podemos retornar funciones que computan mónadas.

Por último, `mfix` es una versión general de la recursión monádica, esta puede tomar una cantidad arbitraria de argumentos. Está motivada por la limitación de `mfix1`, `mfix2` y `mfix3`. Aún así, su uso no es estándar porque todavía está en desarrollo.

3.3 EL COSTO DE LA MÓNADA

Como mencionamos anteriormente, las funcionalidades de MTAC2 tienen un coste. Imaginemos que estamos calculando el cociente entre dos números enteros, siendo el divisor igual a cero. En este caso, el programa no puede calcular el cociente y debe fallar. Esto nos muestra la gran diferencia entre un programa de Coq y uno de MTAC2: un programa monádico puede fallar. Mientras tanto, en el mundo de Coq este concepto no existe. Un programa que retorna un número entero, debe retornar un entero y, más aún, un programa que tiene el tipo de una proposición, efectivamente es una prueba de la misma. Supongamos esa proposición es `P : Prop`. Ahora, para probarla monádicamente necesitamos un programa `p : M P`, pero para cualquier `P` podemos escribir dar un programa con esa signatura y que no sea una prueba.

```
Definition univ (P : Prop): M P :=
  raise MyException.
```

Con `raise` podemos levantar una excepción, en este caso `MyException` es una excepción definida por nosotros. Sin la presencia de la mónada esto no sería posible. La mónada nos quita garantías que, si no, tendríamos en Coq.

Dada esta limitación, todas las funciones nativas de Coq pueden ser utilizadas en los tipos de las funciones, pero no sucede lo mismo con las funciones monádicas. Esto hace que el desarrollo de metaprogramas se tenga que pensar de manera estratégica: qué funciones serán monádicas y cuales serán nativas de Coq.

3.4 ALTERNATIVAS

MTAC2 no es el único metalenguaje de programación para Coq. Un ejemplo de otro metalenguaje es *Ltac2* [**Ltac2**]. Este funciona como un *wrapper* alrededor del *proof engine* de Coq. Implementa una α táctica de tipo monádico en OCaml y trata de conservar la mayor parte de Ltac posible, busca la mayor retrocompatibilidad. Otras alternativas pueden ser Template-Coq [**DBLP:conf/itp/AnandBCST18**], Rtac [**DBLP:conf/esop/MalechaB16**] y Coq-Elpi [**tassi:hal-01637063**]. La gran diferencia entre estos metalenguajes es que MTAC2 es *shallowly embedded*, mientras que los otros no. A diferencia de MTAC2, los objetos de estas alternativas son términos de un tipo inductivo `Term`, o sea, no tienen un tipo informativo. Es decir, en MTAC2 un programa `M A` está garantizado que, si termina correctamente, retorna un término de tipo `A`. Sin embargo, en los lenguajes *deeply embedded*, como los mencionados, nada garantiza que el término retornado tenga un tipo dado, puede retornarse cualquier elemento de cualquier tipo, incluso términos mal construidos.

3.5 TELESCOPIOS

Los *telescopios* son una estructura de datos inductiva que permite expresar una secuencia de tipos o valores, posiblemente dependientes, y de largo arbitrario. Los telescopios, junto con las funciones que lo acompañan, serán claves a la hora de poder expresar nuestro problema y consecuente solución.

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

Listing 3.1: Definición de telescopio

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.

Definiremos la siguiente notación para poder referirnos a los telescopios de una manera más accesible. El constructor `mBase` representa el telescopio vacío o de largo cero. Representaremos con $[\ ]_t$ a `mBase`.

Para un telescopio de largo n, tenemos al constructor `mTele` anidado n veces. Dado que podemos pensar al telescopio como una secuencia dependiente de tipos, un posible comienzo es $[T_0 : \texttt{Type} ;> \ldots]_t$. Ahora, todos argumentos siguientes del telescopio pueden depender de T_0. Luego, $[T_0 : \texttt{Type} ;> T_1 : T_0 \to \texttt{Type} ;> \ldots]_t$ tiene sentido, y el tipo T_1 depende de T_0. Notar que podría ocurrir que ningún argumento dependa de T_0.

```
(* ej 1 *) mBase ≡ [ ]t
(* ej 2 *) mTele (fun T : Type ⇒ mTele R : T → Type) ≡
[T : Type ;> R : (T → Type)]t
(* ej 3 *) mTele (fun T : Type ⇒ mTele t : T) ≡
[T : Type ;> t : T]t
```

Listing 3.2: Notación de telescopios

Para comenzar a estudiar a este tipo, podemos pensar que existen jerarquías. La primera jerarquía sería la de los telescopios en sí, elementos de tipo `MTele`. Ahora, dado un `m : MTele`, la segunda jerarquía es la de los tipos generados por el telescopio `m`. Luego, el último nivel es el de los elementos de este nuevo tipo telescopico.

Veremos esto en mayor profundidad, pero, para eso, debemos primero definir el tipo `Sort`.

```
Inductive Sort : Type := Props | Types .
```

Listing 3.3: Definición de `Sort`

Más adelante en esta sección, utilizaremos `Sort` para abstraer el concepto de tipo y proposición, y poder aplicar ciertas funciones telescópicas.

- En el nivel superior, definimos nuestro telescopio `m` de largo n con dependencias arbitrarias. Esto es simplemente utilizando los constructores de `MTele`.
- Ya con `m` definido, podemos utilizar la función `MTele_Sort` para computar un tipo derivado del telescopio. Sea `(s : Sort)`, es decir, `s` es $\texttt{Type}_s$ o $\texttt{Prop}_s$, la expresión `(MTele_Sort s m)` es un tipo y, específicamente, es el tipo $\forall\ \texttt{x}_1\ \texttt{x}_2\ \ldots\ \texttt{x}_n\texttt{, s}$.
- Como dijimos, `(MTele_Sort s m)` es un tipo, por lo tanto, podemos tener elementos de ese tipo. Esta sería la última jerarquía.

  Para esto, utilizamos la función `MTele_val`. Esta función toma un valor de `(MTele_Sort s m)` y retorna un valor de tipo `s`. Por cómo es el tipo, se deduce que un valor del mismo es algo de la forma $(\texttt{fun}\ \texttt{x}_1\ \texttt{x}_2\ \ldots\ \texttt{x}_n \Rightarrow \texttt{T}\ \texttt{x}_1\ \texttt{x}_2\ \ldots\ \texttt{x}_n)$ para algún `T`. Luego, `MTele_val` retornará un valor de tipo `s`, es decir, $\texttt{T}\ \texttt{y}_1\ \texttt{y}_2\ \ldots\ \texttt{y}_n$ para algunos $\texttt{y}_i$.

Utilizaremos `MTele_Ty` y `MTele_Prop` para expresar `MTele_Sort` $\texttt{Type}_s$ y `MTele_Sort` $\texttt{Prop}_s$ respectivamente.

3.5.1 *Funciones extra*

Los telescopios de MTAC2 traen consigo muchas funciones que son las que le dan el poder expresivo que los hace útiles.

La función `MTele_C` permite mapear a un `MTele_Sort` con una función constante. Por ejemplo, la mónada que hemos observado antes: `M`, en realidad es una función de `Type` en `Prop`. Entonces, podríamos tomar un tipo $\forall\ \texttt{x}_1\ \ldots\ \texttt{x}_n\texttt{, T x}_1\ \ldots\ \texttt{x}_n$ y transformalo en $\forall\ \texttt{x}_1\ \ldots\ \texttt{x}_n\texttt{, M (T x}_1\ \ldots\ \texttt{x}_n\texttt{)}$

Con `MTele_In`, podemos ganar acceso a múltiples tipos y valores telescopios al mismo tiempo, siendo así capaces de computar un nuevo tipo telescópico. Al momento de utilizarlo lo estudiaremos en más profundidad.

3.5.2 *Ejemplo*

Para que los conceptos queden claros, vamos a plantear un ejemplo de telescopios.

Primero, definimos un telescopio `m`.

```
Definition m := [T : Type ;> l : list T ;> p : length l = 0]_t
```

El telescopio tiene tres elementos: el tipo `T`, una lista de tipo `T` (que ya hemos introducido, por lo tanto podemos utilizar) y una prueba `p` de que la lista `l` tiene largo cero.

Si ahora inspeccionamos quien es `MTele_Sort` $\texttt{Type}_s$ `m`, veremos que es lo esperado.

```
Eval cbn in MTele_Ty m.
= ∀ (T : Type) (l : list T), length l = 0 → Type : Type
```

El próximo paso es hablar de elementos de este tipo.

```
Definition Tm : MTele_Ty m := fun T l p ⇒ l = nil.
```

Donde tenemos `l = nil` podemos poner cualquier tipo. Podríamos tener `nat` pero, para hacerlo más interesante definimos un tipo que contenga algún significado lógico. Lo importante es que Coq acepta la definición de `Tm`, lo que implica que esta definición efectivamente tiene el tipo esperado. Si este no fuera el caso, el type-checker de Coq no lo aceptaría. El próximo paso es definir un elemento del tipo `Tm`.

Podemos definir el siguiente teorema y probarlo.

```
Definition test_Tm : forall T (l : list T) (p : List.length l =
    0), l = nil.
intros T l p. by apply length_zero_iff_nil.
Qed.
```

El truco es que, al conocer `Tm` sabemos que esta función tiene ese tipo. Finalmente, podemos definir lo siguiente.

```
Definition vm : MTele_val tm := test_Tm.
```

Efectivamente, `test_Tm` tiene tipo `MTele_val tm`.

Parte II

LIFT

4

MOTIVACIÓN

Mtac2 nos permite definir funciones monádicas. Estas cuentan con ciertas ventajas. La siguiente función calcula el máximo de una lista de números `nat : Set` y utiliza `mtmmatch` para hacer un análisis sintáctico, a diferencia de `match`.

```
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] ⇒ fun P ⇒ M.ret e
  | [? e1 e2 l'] (e1 :: e2 :: l') ⇒ fun P ⇒
   let x := Nat.max e1 e2 in
   f (x :: l') _
  | [? l' r'] app l' r' ⇒ fun P ⇒
   match app_not_nil l' r' P with
   | inl P' ⇒ f l' P'
   | inr P' ⇒ f r' P'
   end
  end.
```

Listing 4.1: Función `list_max_nat`

El último caso de `mtmmatch` analiza una expresión que no es un constructor del tipo inductivo `list`. Por eso, es necesario utilizar el `match` monádico. Notar que tampoco podemos utilizar `mmatch` ya que deseamos retornar términos de tipo `l' <> nil → M nat`.

Ahora, supongamos que deseamos parametrizar $\mathbb{N}$ y utilizar una función que acepte múltiples conjuntos de nuestro interés.

```
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.
```

Listing 4.2: Función `max`

Sea `max` en 4.2, la función que retorna la relación máximo en un conjunto S. A primera vista, nuestra idea podría funcionar, es decir, intuitivamente no es ilógica.

```
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] ⇒ fun P ⇒ M.ret e
  | [? e1 e2 l'] (e1 :: e2 :: l') ⇒ fun P ⇒
   m ← max e1 e2;
```

```
    f (m :: l') _
  | ...
  end.
```

Listing 4.3: Función `list_max`

Al intentar que Coq interprete la función veremos que esta función no tipa. Esto es debido a que `bind` no tiene la signatura necesaria.

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

Nuestro `mfix` no puede unificarse a `M B`, ya que su tipo es una implicación: `(f : forall (l: list S), l' <> nil → M S)`.

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 operadores.

En nuestra solución, suplantamos `bind` y `ret` por `bind^` y `ret^`, respectivamente. Llamaremos a estas funciones las versiones *lifteadas* de las originales. Estas poseen el tipo necesario.

Nuestro proyecto es la codificación de un nuevo metaprograma Lift que puede generalizar metaprogramas con las dependencias necesarias para que sea utilizado en el contexto. En nuestro ejemplo, utilizando Lift podemos generalizar `bind` y `ret`, consiguiendo nuevos metaprogramas que se comportan como los originales pero con una signatura distinta, permitiendo su uso.

```
Definition list_max_lifted (S: Set) :=
max <^- max S; (* notacion para bind^ *)
mfix f (l: list S) : l <> nil → M S :=
   mtmmatch l as l' return l' <> nil → M S with
   | [? e] [e] ⇒ ret^ (fun _ ⇒ e)
   | [? e1 e2 l'] (e1 :: e2 :: l') ⇒ fun P ⇒
    m ← max e1 e2;
    f (m :: l') _
   | ...
   end.
```

Listing 4.4: Función `list_max` lifteada

Veremos que la función Lift requiere pequeño esfuerzo para ser utilizada.

5

LIFT

Denominamos Lift al metaprograma desarrollado en este trabajo. Esta función tiene la tarea de agregar dependencias a otros metaprogramas de manera cuasi automática. Veremos que solo depende de un telescopio, el que, en principio, generaremos nosotros.

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 metaprogramas más generales. Cuando digamos "dependencias triviales" nos referiremos a valores en el tipo que son dependientes pero no tienen ningún valor real, es decir, no influyen en el resultado de la ejecución. Aunque esto pueda parecer inútil, su justificación es que hacer estos agregados, efectivamente genera funciones más generales que pueden ser utilizadas en una amplia cantidad de situaciones.

Esta generalización hace uso de telescopios para poder especificar las dependencias que se desea agregar. Primero, observaremos la situación desde un punto de vista más bien intuitivo y finalizaremos concretizando los telescopios.

5.1 DEFINIENDO OBJETIVOS

Una de las funciones monádicas más simples es `ret`, uno de los operadores monádicos.

```
ret : forall A, A → M A
```

Listing 5.1: Signatura de `ret`

Como vemos en la signatura de `ret`, solo puede devolver expresiones `(M A)` donde no es posible que el retorno esté cuantificado, es decir, no puede retornar una función. Si tomamos `list_max_lifted` 4.4, la función `ret^` toma un valor de tipo `(l' <> nil → S)` y retorna `l' <> nil → M S`, que también puede ser expresado como `forall (p : l' <> nil), M S`. Notemos que, en `list_max_lifted`, el argumento de `ret^` no utiliza el valor introducido en la función.

En principio, el tipo `A : Type` de `ret` será reemplazado por un nuevo tipo que tome la prueba: `(A : l' <> nil → Type)` y naturalmente podemos deducir como queda el resto.

```
ret^ : ∀ (A : l' <> nil → Type), (∀ p, A p) → (∀ p, M (A p))
```

Listing 5.2: Signatura deseada de `ret`

Ahora estudiemos que sucede con `bind`. El operador `bind` es la otra función que debemos modificar, pero su signatura presenta un problema más complejo que `ret`. Esto es debido a que su signatura contiene una función y esta depende de los argumentos `A` y `B`, que se encuentran bajo la mónada en diferentes partes del tipo.

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

Listing 5.3: Signatura de `bind`

Volviendo a `list_max` 4.3, pudimos notar que `bind` no es suficientemente general. En 4.4, observamos que queremos bindear `max S` con nuestro fixpoint. La signatura de `max S` es `M (S → S → S)`. La signatura del fixpoint es `forall (l : list S), l <> nil → M S`, por lo tanto, es lo que debe retornar nuestro `bind`.

Ahora, intentemos armar la signatura del nuevo `bind`. Tenemos que los tipos `A` y `B` que toma `bind` deben ser reemplazados por otros más generales. Si hacemos que estos dos tipos sean dependientes de los mismos argumentos, podemos facilitar este razonamiento. Entonces, si `(A B : ∀ l (p : l <> nil), Type)`, podemos reescribir la signatura.

```
bind^ : ∀ (A B : ∀ l (p : l <> nil), Type),
  ∀ l p, M (A l p) →
  ∀ l p, (A l p → M (B l p)) →
  ∀ l p, M (B l p)
```

Listing 5.4: Signatura deseada de `bind`

En esta signatura, podemos notar que el retorno de `bind^` es, efectivamente, el tipo del punto fijo. También, notamos que la función de `A` en `M B`, intuitivamente sigue siendo eso, solo que tomamos dos argumentos `l` y `p` los cuales `A` y `B` están compartiendo. Ahora, es importante que, al momento de utilizar `bind^`, los dos tipos que toma estén bien definidos. Necesitamos que `(B l p)` reduzca a `S` y que `(A l p)` reduzca a `(S → S → S)`.

5.2 CREANDO TELESCOPIOS

Como vimos anteriormente, para liftear una función necesitamos definir un telescopio. Este es el que indicará los argumentos dependientes que debamos agregar a nuestra función.

Como vimos recién, para `bind^`, en el caso de 4.4, debemos añadir dos dependecias a los tipos: `l : list S` y `p : l <> nil`. Por ese motivo, necesitaremos un telescopio de largo dos. Sin embargo, dado que `ret^` es utilizado dentro del fixpoint, `l` ya se encuentra en el contexto y no es necesario añadirlo como un dato al telescopio. Luego, necesitaremos dos telescopios.

```
m : MTele := [l : list S ;> p : l <> nil]t
n : MTele := fun l ⇒ [p : l <> nil]t
```

Listing 5.5: Telescopio para `list_max`

No es un problema que `n` dependa de `l`. Podemos crear una lista cualquiera al momento de liftear.

En la sección 7.2, discutiremos sobre la capacidad de generar estos telescopios de manera automática, analizando el tipo deseado de nuestra función lifteada.

5.3 EL RESULTADO

La función LIFT retorna una tupla dependiente con el tipo de la función lifteada y la función lifteada. Con nuestro telescopio `m`, ahora extraer la nueva función es simple.

`mprojT2` nos permite extraer el segundo argumento de una tupla dependiente en MTAC2

```
bind^ := mprojT2 (lift bind m)
ret^ := mprojT2 (lift ret (n l))
```

Listing 5.6: Lifteando `ret` y `bind`

Finalmente, nuestro programa quedaría de la siguiente forma.

```
Definition list_max_lifted (S: Set) :=
max <^- max S; (* notacion para bind^ *)
mfix f (l: list S) : l' <> nil → M S :=
   mtmmatch l as l' return l' <> nil → M S with
   | [? e] [e] ⇒ ret^ (fun _ ⇒ e)
   | [? e1 e2 l'] (e1 :: e2 :: l') ⇒ fun P ⇒
     m ← max e1 e2;
     f (m :: l') _
   | [? l' r'] app l' r' ⇒ fun P ⇒
     match app_not_nil l' r' P with
     | inl P' ⇒ f l' P'
     | inr P' ⇒ f r' P'
     end
   end.
```

Listing 5.7: Versión final de `list_max`

En el próximo capítulo, daremos la definición precisa de LIFT, en la sección 6.3, y veremos aspectos técnicos de la misma. El código de la función se encuentra en el apéndice A.

6 DEFINICIÓN DE LIFT

En este capítulo, mostraremos en detalles las herramientas necesarias para la definición de Lift, la cual se hará en la sección 6.3. Su código se encuentra en el apéndice A.

6.1 EL TIPO 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.

Los nombres de los constructores están simplificados para la facilidad del lector. En el apéndice podemos encontrar la verdadera definición.

```
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
.
```

Listing 6.1: El tipo inductivo `TyTree`

Notemos que el constructor `M` es distinto a `M`. En este trabajo, podemos diferenciarlos por el color. Veremos que ambos representan lo mismo, solo que uno es la versión reflejada del otro.

Con los constructores de este tipo podemos expresar todas las signaturas que nos interesan. Varios de los constructores, como por ejemplo `MFA` o `FATeleVal`, tendrán sentido más adelante.

Ahora, veamos algunos ejemplos simples de cómo podemos reescribir signaturas con este nuevo tipo.

```
ret : ∀ A, A → M A.
ret_r : FAType (fun A ⇒ imp (base A) (M A)).
bind : ∀ A B, M A → (A → M B) → M B.
bind_r : FAType (fun A ⇒ FAType (fun B ⇒ imp (M A) (imp (imp (base
     A) (M B)) (M B)))).
f : ∀ (n : nat), 0 <= n → M nat.
```

```
f_r : FAVal nat (fun n ⇒ imp (base (0 <= n)) (M nat)).
```

Listing 6.2: Ejemplos de TyTree

A primera vista, los tipos pueden escribirse de múltiples formas ya que FAVal es más fuerte que imp. La clave está en que nuestra función Lift hará una separación muy específica entre cada caso, por lo tanto, la forma en que reescribamos la signatura de las funciones es sumamente importante.

Utilizaremos 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 utilizarla en las signaturas que definimos. En cambio, la función to_tree : Type → M TyTree, que es análoga a to_ty, necesariamente será monádica ya que debemos hacer un análisis sintáctico en los tipos de Coq.

Se puede encontrar la definición de to_ty y to_tree en el apéndice A.

6.2 TYTREES MONÁDICOS

En esta sección, 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, donde la función está parametrizada por un telescopio m. Esto significa que podemos liftear una función sin un telescopio m predefinido y luego instanciarlo.

```
fun m : MTele ⇒
 FATeleType m
  (fun A : MTele_Ty m ⇒
   imp (In Type_s (fun a : accessor m ⇒ acc_sort a A))
   (MFA A))
```

Listing 6.3: ret lifteado

Podemos encontrar en rojo todos los constructores de TyTree que se utilizan. A continuación, definiremos todos los elementos desconocidos de esta signatura presentada.

Con FATeleType podemos introducir tipos telescopios, es la versión telescopica de FAType. Esto signfica que en la signatura de la función, si FAType introduce un tipo que se encuentra bajo la mónada, este constructor se reemplazará por FATeleType. En caso contrario, no debemos reemplazar este constructor en la signatura.

La función 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.

Un accessor es un par de funciones llamadas acc_sort y acc_val. Se comportan de la siguiente forma:

- `acc_sort` convierte `MTele_Sort s n` en `stype_of s`.
- `acc_val` convierte `MTele_val T` en `acc_sort T`.

Con la función `acc_sort`, podemos transformar el tipo telescopico `MTele_Sort s n` en `stype_of s`, es decir, en `Type` o en `Prop` si `s` es $\texttt{Type}_s$ o `Prop_sort`, respectivamente. Esto representa añadir al contexto todas las depedencias y referirnos al tipo base. Mientras tanto, con la segunda función, conseguimos un elemento de ese tipo que `acc_sort` nos concede. Intuitivamente, `accessor` nos permite "acceder" al tipo con todos los argumentos ya presentes en el contexto.

Utilizamos `MFA` para representar tipos monádicos cuantificados.

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

Listing 6.4: Definición de `MFA`

Con `MFA` representamos tipos monádicos con argumentos cuantificados. Sea `(t : MTele)` de largo n y `(T : MTele_Ty t)`, `MFA T` representará $\forall$ `x`$_1$ `...` `x`$_n$`, M (T x`$_1$ `...` `x`$_n$`)`.

En el caso del Listing 6.3, la signatura `(In` $\texttt{Type}_s$ `(fun a : accessor m ⇒ acc_sort a A))` es simplemente equivalente a `(val 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 similar a la matemática y la función resultante será muy simple. Por eso, utilizemos el telescopio `n` de nuestra motivación, situado en el Listing 5.5.

```
ret^ : ∀ A : l <> nil → Type,
  (∀  p : l <> nil, A p) →
  ∀ p : l <> nil, M (A p) :=
  fun A x p ⇒ ret (x p)
```

Listing 6.5: Ejemplo de `ret^`

Notemos que esta solución efectivamente se puede utilizar en la función del Listing 4.4. Podemos observar también la signatura en forma `TyTree` y notar que se corresponde con lo esperado.

```
FATeleType n
  (fun A : l <> nil → Type ⇒
    imp (In Type_s (fun a : accessor n ⇒ acc_sort a A))
      (MFA A))
```

Listing 6.6: Signatura en `TyTree` de `ret^`

Si concretizamos el telescopio para `bind^`, tendremos el siguiente resultado.

```
bind^ : ∀ A B : ∀ l : list nat, l <> nil → Type,
  (∀ (l : list nat) (p : l <> nil), M (A l p)) →
  (∀ (l : list nat) (p : l <> nil), A l p → M (B l p)) →
  ∀ (l : list nat) (p : l <> nil), M (B l p) :=
  fun A B x f y ⇒ bind (x l p) (f l p)
```

Listing 6.7: Ejemplo de `bind^`

El único constructor que no hemos utilizado es `FATeleVal`. Este cumple el mismo rol que `FAVal` y, como ya viene siendo el caso, el constructor solo será reemplazado si el tipo del elemento que introduce está bajo la mónada.

6.3 EL ALGORITMO

Aquí definiremos y analizaremos la función LIFT.

```
Fixpoint lift (m : MTele) (U : ArgsOf m) (T : TyTree) :
  ∀ (f : to_ty T), M m:{ T : TyTree & to_ty T}.
```

Listing 6.8: Signatura de LIFT

Dentro de esta signatura tenemos múltiples elementos conocidos:

- El telescopio `m` que describe las dependencias a agregar.
- El TyTree `T` con la signatura de la función.
- La función `f : to_ty T`. La función a liftear.

LIFT retorna un par dependiente (o Σ-type) que contiene la nueva función junto con el `TyTree` que describe su tipo.

El argumento restante es `U : ArgsOf m`.

```
Fixpoint ArgsOf (m : MTele) : Type :=
  match m with
  | mBase ⇒ unit
  | mTele f ⇒ msigT (fun x ⇒ ArgsOf (f x))
  end.
```

Listing 6.9: Definición de `ArgsOf`

Este contiene los argumentos que el telescopio añade de manera descurrificada. Esto quiere decir que transporta los argumentos del telescopio en un "contenedor". Cuando encontramos un tipo `A` cualquiera en nuestra función, este tipo puede o no estar bajo la mónada. Si no lo está, no debemos realizar ninguna tarea. En el caso contrario, debemos modificarlo. Esto signfica reemplazar al tipo original `A` por `A : MTele_Ty t` con `t : MTele`. Notemos que `A` puede ser cualquier tipo. Con la función `apply_sort`, podemos aplicar a `U` en `A`. La función generará un nuevo tipo: ∀ v_1 ... v_n, `A` v_1 ... v_n.

Cuando en LIFT encontramos una implicación, debemos liftear el lado izquierdo para luego poder liftear el derecho. Para hacer esto, utilizamos la función `lift_in`. A través de otras funciones auxiliares, `lift_in` nos permitirá reemplazar nuestro tipo `apply_sort A U` por un otro tipo equivalente: `F (uncurry_in_acc U)` donde `F := fun a ⇒ a.(acc_sort)A`. La función `uncurry_in_acc` toma nuestro `U : ArgsOf m` y nos devuelve un `accessor m`. De esta forma, `F (uncurry_in_acc U)` se traduce a `(uncurry_in_acc U).(acc_sort)A`, lo que representa el tipo `A` con los argumentos de `U`.

Necesitamos codificarlo de esta forma para que sea aceptado por `MTele_In` y, además, `lift_in` nos provee del tipo junto con una prueba de equivalencia entre ambos. De esta forma, podemos reemplazarlo sin problema y podemos liftear así al elemento de la izquierda de la implicación.

6.3.1 *Caso de estudio: `ret`*

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

1. Matcheamos `FAType F` donde `F : Type → TyTree`. Generamos un tipo arbritario `A` para, entonces, poder verificar si, en `F A`, `A` se encuentra bajo la mónada. Realizamos esta verificación con la función auxiliar `is_m`. Esta función retornará `true` ya que efectivamente el tipo `A` se encuentra bajo la mónada. Entonces, generamos un nuevo `A : MTele_Ty t`, es decir, una versión telescopica de `A`. Utilizamos `MTele_Ty` como notación para `MTele_Sort` $\texttt{Type}_s$. Recordemos que `MTele_Sort` se utiliza para referirse a tipos que tengan las depedencias del telescopio.

   Luego, aplicamos LIFT de manera recursiva sobre `F (apply_sort A U)`. Aquí utilizamos la función `apply_sort` de manera de aplicar los argumentos de `U` en el tipo `A`.

2. Ahora, debemos liftear algo del siguiente tipo.

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

   Nuestra expresión matcheará con el caso `imp` de LIFT. Dado que ya hemos introducido todos los tipos cuantificados, sabemos cómo tratar a cada uno. En este caso, eso es muy importante dado que se realiza un chequeo para saber si el lado izquierdo de la implicación contiene un tipo telescópico. Si no lo hay, el lado izquierdo de la implicación será "final", es decir, no necesitará más modificaciones. Pero, en nuestro caso, reemplazamos `A` por `apply_sort A U`. Realizamos este chequeo con la función auxiliar `contains_u`. Esto nos lleva a tener que utilizar `lift_in`.

3. La función `lift_in` se utiliza para liftear argumentos que se encuentran a la izquierda de una implicación. A través de múl-

tiples funciones auxiliares, `lift_in` nos permitirá reemplazar `apply_sort A U` por un tipo equivalente: `F (uncurry_in_acc U)`. La función `F` utilizará a `uncurry_in_acc U` para "acceder" al tipo. Un `accessor` nos permite expresar el "tener acceso" a los valores para cada argumento del telescopio. Podemos pensar que todo esto es la concretización del lifteo del lado izquierdo de la implicación, donde conseguimos esta función `F`. Esto nos es útil porque ahora podemos generar un valor `x : X'` en LIFT, donde `X'` es `MTele_val (MTele_In Type_s F)`. Es decir, `x` es una variable del tipo resultante de liftear `X` en el lado izquierdo de la implicación. Lo que resta es tomar nuestra función `f` de tipo `X' → Y` y liftearla. Esto signfica liftear `f x`.

4. El último paso es ejecutar `lift t U Y (f x)`, ya sabiendo que `Y = M (apply_sort A U)`. Este caso es particular porque no tiene llamadas recursivas. Primero, debemos abstraer a `U` de `f` obteniendo una función dependiente de esta variable. Luego, currificamos a `f` con respecto a `U`. Esto se realiza a través de la función `curry` de MTAC2 y, de esta manera, la función pasa a tener tipo `to_ty (MFA A)`.

5. Finalmente, LIFT retorna un `T' : TyTree` y `ret^ : to_ty (T')` con `T' = FATeleType (fun A ⇒ imp (val A) (MFA A))`.

6.3.2 *Caso de estudio: `bind`*

A continuación, haremos un recorrido paso a paso de cómo es lifteado `bind`.

1. Matcheamos `FAType F` donde `F : Type → TyTree`. Generamos un tipo arbritario `A` para entonces poder verificar si, en `F A`, `A` se encuentra bajo la mónada. Realizamos esta verificación con la función auxiliar `is_m`. Esta función retornará `true` ya que efectivamente el tipo `A` se encuentra bajo la mónada. Entonces, generamos un nuevo `A : MTele_Ty t`, es decir, una versión telescopica de `A`. Utilizamos `MTele_Ty` como notación para `MTele_Sort Type_s` . Recordemos que `MTele_Sort` se utiliza para referirse a tipos que tengan las depedencias del telescopio.

   Luego, aplicamos LIFT de manera recursiva sobre `F (apply_sort A U)`. Aquí utilizamos la función `apply_sort` de manera de aplicar los argumentos de `U` en el tipo `A`.

   Dado que `bind` introduce dos tipos (`A` y `B`), repetimos el paso anterior para reemplazar al tipo `B` cualquiera, por uno telescopico.

2. Ahora nos queda: `M A → (A → M B) → M B`. Dado que esto es una implicación matcheamos el caso de `imp`, y llamamos a `lift_in` en el lado izquierdo.

3. Tenemos que liftear `M A` con `lift_in`. Esto será muy similar al caso de que vemos en `ret`, con la diferencia de que la función `F` que se utiliza en `lift_in` es de la siguiente forma: `F := fun a ⇒ M (a.(acc_sort)A)`. Terminamos reemplazando `baseA` por `InType`$_s$ `(fun a : accessor m ⇒ M (acc_sort a A))`. Recordemos que esto termina siendo equivalente a `MFAA`.

4. A continuación, podemos utilizar el lado izquierdo ya lifteado para liftear el lado derecho. Como el lado derecho también es una implicación, volvemos a correr `lift_in` pero ahora el argumento es de tipo `(A → M B)`, es decir, una función.

5. Al correr `lift_in`, matcheamos el caso de la implicación. La diferencia también es la función `F`. Sea `X := A` y `Y := M B`. Entonces, `F` se define de la siguiente forma: `F := fun a ⇒ FX a → FY a`, donde `FX` y `FY`, son el resultado de ejecutar `lift_in` sobre `X` e `Y` respectivamente.

6. Finalmente, ya hemos calculado el lado el lado izquierdo de la implicación, nos queda liftear el lado derecho: `M B`. Como en `ret`, abstraemos a `U` de `f` obteniendo una función dependiente de esta variable y currificamos a `f` con respecto a `U`.

   Terminamos con un `MFA B` y finalmente el tipo de `bind^` queda de la siguiente forma.

```
FATeleType m
 (λ A : l <> nil → Type ⇒
  FATeleType m
   (λ B : l <> nil → Type ⇒
    imp
     (In Types (λ a : accessor m ⇒ M (acc_sort a A)))
     (imp
      (In Types
       (λ a : accessor m ⇒
        acc_sort a A → M (acc_sort a B)))
      (MFA B))))
```

Hemos visto dos aplicaciones paso a paso de la función Lift, que nos permiten generar las versiones generalizadas de `ret` y `bind` tal como las necesitamos para nuestro ejemplo del Listing 4.4.

Aquí concluimos la discusión técnica, pero antes reiteramos que en el Apéndice A se encuentra a disposición del lector el código de la función Lift.

Parte III

CONCLUSIÓN

7 CONCLUSIONES Y TRABAJO FUTURO

7.1 CONCLUSIÓN

En la Parte i, estudiamos al asistente de pruebas Coq y el metalenguaje de programación Mtac2. Vimos cómo los tipos dependientes son sumamente importantes y cómo en Mtac2 se utilizan mónadas para escribir sus metaprogramas.

Luego, en la Parte ii, comprendimos el problema que se puede generar al utilizar los operadores monádicos de Mtac2 y un uso fuerte de tipos dependientes. Analizamos las signaturas de las funciones y se concluyó en que se necesitaban nuevas funciones más generales. Consecuentemente, desarrollamos el metaprograma Lift que puede generalizar otros metaprogramas de manera casi automática. Para hacer esto, realizamos un análisis caso por caso sobre la signatura de las funciones, y utilizamos telescopios para expresar los argumentos dependientes que deseamos agregar.

Finalmente, en la Sección 6.3 pudimos generalizar las funciones `bind` y `ret`. Estas generalizaciones mostraron ser independientes del telescopio que utilizamos y, por lo tanto, observamos que esta solución es efectivamente más general que la propuesta de específicamente definir nuevas versiones de estos operadores monádicos. Con las generalizaciones, `bind^` y `ret^`, logramos codificar la función `list_max` del Capítulo 4 como lo deseamos en un principio. El único costo fue el de definir los telescopios apropiados para la situación.

7.2 TRABAJO FUTURO

En algún punto se planea añadir Lift a Mtac2 como un feature por defecto. Pero, principalmente, el interés cae en implementar notación inteligente que pueda deducir telescopios. Como vimos en 4, las funciones `bind` y `ret` son funciones que queremos liftear, y probablemente con mayor frecuencia que otras. Con Coq podemos inferir el tipo necesario de estos dos operadores (y otros) y, de esta manera, podemos generar los telescopios, es decir, liftear funciones de manera completamente automática.

A través de la notación, podemos activar un algoritmo por detrás que hará esta inferencia y generará el telescopio. La notación es la siguiente.

```
(* bind de b en c pero con un bind lifteado *)
(* suponemos que a aparece en c *)
a <^- b;
```

```
c
(* equivalente a la notacion de arriba pero *)
(* el resultado de b no influye en c, lo ignoramos *)
b;^;c
(* funcion cualquiera lifteada *)
(* caso de ret^ en la motivacion *)
g^
```

Parte de esta notación ya ha sido desarrollada. La complejidad radica en poder obtener el tipo adecuado en el momento justo para poder generar el telescopio correspondiente y poder liftear la función.

Parte IV

APÉNDICE

APENDICE

Utilizaremos este apéndice para incluir código de la tésis, incluido Lift. Esto es un dump del archivo que contiene a Lift junto con algunos de los ejemplos que utilizamos para la motivación de esta tésis.

El código también puede encontrarse en el siguiente repositorio.

```
From Mtac2 Require Import Base Mtac2 Specif Sorts MTele MFixDef
    MTeleMatch.
Require Import Coq.Lists.List.
Import Sorts.S.
Import M.notations.
Import M.M.

Set Universe Polymorphism.
Unset Universe Minimization ToSet.
Unset Printing Universes.

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

(* If recursion is needed then it's TyTree, if not only Type *)
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 : MTele_val T
    → TyTree) : TyTree
| tyTree_FATeleType (m : MTele) (F : MTele_Ty m → TyTree) :
    TyTree
| tyTree_FAVal (T : Type) (F : T → TyTree) : TyTree
| tyTree_FAType (F : Type → TyTree) : TyTree
| tyTree_base (T : Type) : TyTree
.

Fixpoint to_ty (X : TyTree) : Type :=
 match X as X' with
  | tyTree_val T ⇒ MTele_val T
  | tyTree_M T ⇒ M T
  | tyTree_MFA T ⇒ MFA T
  | tyTree_In s F ⇒ MTele_val (MTele_In s F)
  | tyTree_imp T R ⇒ to_ty T → to_ty R
  | ·tyTree_FATeleVal m T F ⇒ forall T, to_ty (F T)
  | tyTree_FATeleType m F ⇒ forall T : (MTele_Ty m), to_ty (F T)
  | tyTree_FAVal T F ⇒ forall t : T, to_ty (F t)
```

```
  | tyTree_FAType F ⇒ forall T : Type, to_ty (F T)
  | tyTree_base T ⇒ T
  end.

(* This function is not used at all, hence the reason why it's
    not polished *)
Definition to_tree (X : Type) : M TyTree :=
  (mfix1 rec (X : Type) : M TyTree :=
    mmatch X as X return M TyTree with
    | [? T : Type] (M T):Type ⇒
      ret (tyTree_M T)
    | [? T R : Type] T → R ⇒
    (* no dependency of T on R. It's equivalent to forall _ : T, R
        *)
      T ← rec T;
      R ← rec R;
      ret (tyTree_imp T R)
    | [? F : Type → Type] forall T : Type, F T ⇒
      \nu T : Type,
        F ← rec (F T);
        F ← abs_fun T F;
        ret (tyTree_FAType F)
    | [? T (F : forall t : T, Type)] forall t : T, F t ⇒
      \nu t : T,
        F ← rec (F t);
        F ← abs_fun t F;
        ret (tyTree_FAVal T F)
    | _ ⇒ ret (tyTree_base X)
    end) X.

(*** Is-M *)

(* Checks if a given type A is found "under M" *)
(* true iff A is "under M", false otherwise *)
Definition is_m (T : TyTree) (A : Type) : M bool :=
  print "is_m on T:";;
  print_term T;;
  (mfix1 f (T : TyTree) : M bool :=
  mmatch T return M bool with
  | [? X] tyTree_base X ⇒ ret false
  | [? X] tyTree_M X ⇒
    mmatch X return M bool with
    | A ⇒ ret true
    | _ ⇒ ret false
    end
  | [? X Y] tyTree_imp X Y ⇒
    fX ← f X;
    fY ← f Y;
    let r := orb fX fY in
    ret r
  | [? F] tyTree_FAType F ⇒
    print_term (F A);;
```

```
87    \nu X : Type,
88    f (F X)
89   | [? X F] tyTree_FAVal X F ⇒
90    \nu x : X,
91     f (F x)
92   | _ ⇒ ret false
93   end) T.
94
95  (* This function is used to determine if a TyTree contains a
       mention of an element U. The idea is to abstract and if the
       abstraction fails, it means that U is in T. *)
96  Definition contains_u (m : MTele) (U : ArgsOf m) (T : TyTree) : M
        bool :=
97   mtry
98    T' ← abs_fun U T;
99    print "T' on contains_u:";;
100   print_term T';;
101   mmatch T' with
102   | [? T''] (fun _ ⇒ T'') ⇒
103    ret false
104   | _ ⇒
105    ret true
106   end
107  with AbsDependencyError ⇒
108   ret true
109  end.
110
111 (*** Lift In section *)
112
113 Fixpoint uncurry_val {s : Sort} {m : MTele} :
114  forall {A : MTele_Sort s m},
115  MTele_val A → forall U : ArgsOf m, ·apply_sort s m A U :=
116  match m as m return
117      forall A : MTele_Sort s m,
118       MTele_val A → forall U : ArgsOf m, ·apply_sort s m A U
119  with
120  | mBase ⇒ fun A F _ ⇒ F
121  | mTele f ⇒ fun A F '(mexistT _ x U) ⇒ ·uncurry_val s (f x) _ (
        App F x) _
122  end.
123
124 Definition uncurry_in_acc {m : MTele} (U : ArgsOf m) : accessor m
       :=
125  let now_const := fun (s : Sort) (T : s) (ms : MTele_Const T m)
       ⇒ apply_const ms U in
126  let now_val := fun (s : Sort) (ms : MTele_Sort s m) (mv :
       MTele_val ms) ⇒ uncurry_val mv U in
127  Accessor _ now_const now_val.
128
129 Definition uncurry_in {s : Sort} :
130  forall {m : MTele} (F : accessor m → s),
131  (MTele_val (MTele_In s F)) →
```

```
  forall U : ArgsOf m,
    F (uncurry_in_acc U).
  fix IH 1; destruct m; intros.
  + simpl in *. assumption.
  + simpl in *. destruct U. specialize (IH (F x) _ (App X0 x) a).
      assumption.
Defined.

Definition UnLiftInCase : Exception. exact exception. Qed.

(* This is a new type that helps organize the code. *)
Definition lift_inR {m} (T : TyTree) (A : accessor m):=
  m:{F : (accessor m → Type_sort) & (to_ty T = F A)}.

(* This function is an auxiliary function called by lift. It is
    only used for tyTree_imp, for the left side of the
    implication *)
Definition lift_in {m : MTele} (U : ArgsOf m) (T : TyTree) :
             M (lift_inR T (uncurry_in_acc U)) :=
  (mfix1 f (T : TyTree) : M (lift_inR T (uncurry_in_acc U)) :=
    mmatch T as e return M (lift_inR e _) with
    | [? (A : MTele_Ty m)] tyTree_base (apply_sort A U) ⇒
      print "lift_in: base";;
      let F : (accessor m → Type) := fun a ⇒ a.(acc_sort) A in
      let eq_p : to_ty (tyTree_base (apply_sort A U)) = F (
          uncurry_in_acc U) := eq_refl in
      ret (mexistT _ F eq_p)
    | [? (A : MTele_Ty m)] tyTree_M (apply_sort A U) ⇒
      print "lift_in: M";;
      let F : (accessor m → Type) := fun a ⇒ M (a.(acc_sort) A)
          in
      let eq_p : to_ty (tyTree_M (apply_sort A U)) = F (
          uncurry_in_acc U) := eq_refl in
      ret (mexistT _ F eq_p)
    | [? X Y] tyTree_imp X Y ⇒
      print "lift_in: imp";;
      '(mexistT _ FX pX) ← f X;
      '(mexistT _ FY pY) ← f Y;
      let F := fun a ⇒ FX a → FY a in
      let eq_p : to_ty (tyTree_imp X Y) = F (uncurry_in_acc U) :=
        ltac:(simpl in *; rewrite pX, pY; refine eq_refl) in
      ret (mexistT _ F eq_p)
    | _ ⇒ raise UnLiftInCase
    end) T.

(*** Lift section *)

Fixpoint MTele_Cs {s : Sort} (n : MTele) (T : s) : MTele_Sort s n
    :=
  match n as n return MTele_Sort s n with
  | mBase ⇒
    T
```

```
177   | ·mTele X F ⇒
178     fun x : X ⇒ ·MTele_Cs s (F x) T
179   end.
180
181 Fixpoint MTele_cs {s : Sort} {n : MTele} {X : Type} (f : M X) :
        MFA (·MTele_Cs Type_sort n X) :=
182   match n as n return MFA (·MTele_Cs Type_sort n X) with
183   | mBase ⇒
184     f
185   | ·mTele Y F ⇒
186     ·Fun Type_sort Y (fun y : Y ⇒ MFA (·MTele_Cs Type_sort (F y) X
            )) (fun y : Y ⇒ ·MTele_cs s (F y) X f)
187   end.
188
189 (* Next line needs to be after MTele_cs, if not, Coq fails to
        typecheck *)
190 Arguments MTele_Cs {s} {n} _.
191
192 Definition ShitHappens : Exception. exact exception. Qed.
193
194 (* It has a lot of prints for easier debugging *)
195 Polymorphic Fixpoint lift (m : MTele) (U : ArgsOf m) (T : TyTree)
        :
196   forall (f : to_ty T), M m:{ T : TyTree & to_ty T} :=
197   match T as T return forall (f : to_ty T), M m:{ T' : TyTree &
        to_ty T'} with
198   | tyTree_base X ⇒
199     fun f ⇒
200       ret (mexistT (fun Y : TyTree ⇒ to_ty Y) (tyTree_base X) f)
201   | tyTree_M X ⇒
202     fun f ⇒
203       print "lift: M";;
204       mmatch mexistT (fun X : Type ⇒ to_ty (tyTree_M X)) X f
205       return M m:{ T' : TyTree & to_ty T'} with
206       | [? (A : MTele_Ty m) f]
207         mexistT (fun X : Type ⇒ to_ty (tyTree_M X))
208                 (apply_sort A U)
209                 f ⇒
210           print "T:";;
211           print_term (to_ty T);;
212           print "f:";;
213           print_term f;;
214           f ← ·abs_fun (ArgsOf m) (fun U ⇒ to_ty (tyTree_M (
                apply_sort A U))) U f;
215           print "survive2";;
216           let f := curry f in
217           ret (mexistT _ (tyTree_MFA A) f)
218       | _ ⇒
219         (* Constant case *)
220         let T := ·MTele_Cs Type_sort m X in (* okay *)
221         let f' := ·MTele_cs Type_sort m X f in
222         ret (mexistT (fun X : TyTree ⇒ to_ty X) (tyTree_MFA T) f')
```

```
223       end
224   | tyTree_imp X Y ⇒
225     fun f ⇒
226       print "lift: imp";;
227       print "X on imp:";;
228       print_term X;;
229       b ← contains_u m U X;
230       print "b on imp:";;
231       print_term b;;
232       if b then
233         mtry
234           ('(mexistT _ F e) ← lift_in U X;
235           \nu x : MTele_val (MTele_In Type_sort F),
236             (* lift on right side Y *)
237             let G := (F (uncurry_in_acc U)) → to_ty Y in
238             match eq_sym e in _ = T return (T → to_ty Y) → M _ with
239             | eq_refl ⇒ fun f ⇒
240               '(mexistT _ Y' f) ← lift m U (Y) (f (uncurry_in (s:=
                      Type_sort) F x U));
241               f ← abs_fun x f;
242               print "survive1";;
243               ret (mexistT to_ty
244                  (tyTree_imp (tyTree_In Type_sort F) Y')
245                  f)
246           end f)
247         with UnLiftInCase ⇒
248           mfail "UnLiftInCase raised"
249         end
250       else
251         (* Because X does not contain monadic stuff it's assumed it'
                s "final" *)
252         \nu x : to_ty X,
253           '(mexistT _ Y' f) ← lift m U (Y) (f x);
254           f ← abs_fun x f;
255           ret (mexistT to_ty (tyTree_imp X Y') f)
256   | tyTree_FAVal X F ⇒
257     fun f ⇒
258       print "lift: FA";;
259       \nu x : X,
260        '(mexistT _ F f) ← lift m U (F x) (f x);
261        F ← abs_fun x F;
262        f ← coerce f;
263        f ← abs_fun x f;
264        ret (mexistT _ (tyTree_FAVal X F) (f))
265   | tyTree_FAType F ⇒
266     fun f ⇒
267       print "lift: FAType";;
268       \nu A : Type,
269       b ← is_m (F A) A;
270       if b then (* Replace A with a (RETURN A U) *)
271         \nu A : MTele_Ty m,
272           (* I use apply_sort A U to uncurry the values *)
```

```
273         s ← lift m U (F (apply_sort A U)) (f (apply_sort A U));
274         let '(mexistT _ T' f') := s in
275         T'' ← abs_fun (P := fun A ⇒ TyTree) A T';
276         f' ← coerce f';
277         f'' ← abs_fun (P := fun A ⇒ to_ty (T'' A)) A f';
278         let T'' := tyTree_FATeleType m T'' in
279         print "T'':";;
280         print_term (to_ty T'');;
281         print "f'':";;
282         print_term f'';;
283         ret (mexistT to_ty T'' f'')
284       else
285         (* A is not monadic, no replacement *)
286         s ← lift m U (F A) (f A);
287         let '(mexistT _ T' f') := s in
288         T'' ← abs_fun (P := fun A ⇒ TyTree) A T';
289         f' ← coerce f';
290         f'' ← abs_fun (P := fun A ⇒ to_ty (T'' A)) A f';
291         let T'' := tyTree_FAType T'' in
292         print "T'':";;
293         print_term T'';;
294         print "f'':";;
295         print_term f'';;
296         ret (mexistT to_ty T'' f'')
297   | _ ⇒ fun _ ⇒
298     print_term T;;
299     raise ShitHappens
300   end.
301
302 (* For easier usage *)
303 Definition lift' {T : TyTree} (f : to_ty T) : MTele → M m:{T :
        TyTree & to_ty T} :=
304   fun (m : MTele) ⇒
305   \nu U : ArgsOf m,
306     lift m U T f.
307
308 (** Everything works! *)
309
310 (** ret *)
311 (* We lift ret and are interested in using the telescope from the
        motivation *)
312 Definition retTyTree := tyTree_FAType (fun A : Type ⇒ (tyTree_imp
        (tyTree_base A) (tyTree_M A))).
313 (* We have to make an alias to ret so we can tell Coq that we
       want to use a TyTree to refer to it's type. This works
       because retTyTree is effectively equivalent to the original
       type of ret *)
314 Definition rett : to_ty retTyTree := ·ret.
315
316 (* We get the lifted ret in a Sigma-type wrapper. In this case we
        don't define a telescope, it's the more general case *)
```

```
317 Definition l_ret (m : MTele): m:{T : TyTree & to_ty T} := ltac:(
        mrun (lift' rett m)).
318
319 (* General MTele. Our list_max has arguments T and l which are
        know at the moment of interest *)
320 (* For ret *)
321 Definition n_fun := fun (T : Type) (l : list T) ⇒ mTele (fun p :
        l <> nil ⇒ mBase).
322 (* For bind *)
323 Definition m_fun := fun (T : Type) ⇒ mTele (fun l : list T ⇒
        mTele (fun p : l <> nil ⇒ mBase)).
324
325 (* Example list, not relevant *)
326 Definition l : list nat := cons 3 (cons 1 (cons 10 (cons 7 nil))).

327
328 (* We need this proof to create the U : ArgsOf m *)
329 Lemma l_nil : l <> nil.
330 Proof.
331 unfold l. unfold not. intros H. apply eq_sym in H. apply nil_cons
        in H. apply H.
332 Qed.
333
334 (* Final MTele *)
335 Definition m := m_fun nat.
336 Definition n := n_fun nat l.
337 Eval cbn in ArgsOf m.
338 (* We can now define U *)
339 Definition U : ArgsOf n := mexistT _ l_nil tt.
340
341 (* This won't work *)
342 (* Definition li_ret : m:{T : TyTree & to_ty T} := ltac:(mrun (
        lift m U retTyTree ret)). *)
343
344 (* This works *)
345 Definition li_ret : m:{T : TyTree & to_ty T} := ltac:(mrun (lift'
        rett n)).
346
347 (* Check the result of li_ret *)
348 Eval cbn in to_ty (mprojT1 li_ret).
349 Eval cbn in mprojT2 li_ret.
350 Eval cbn in mprojT1 li_ret.
351
352 (* Check the result of l_ret *)
353 Eval cbn in to_ty (mprojT1 (l_ret m)).
354 Eval cbn in mprojT2 (l_ret m).
355
356 (* The result of the previous examples leads to the same result.
        Thus, we conclude that it is not necessary to define the
        telescope before. *)
357
358 (** bind *)
```

```
359
360 About bind.
361 Definition bindTyTree := tyTree_FAType (fun A : Type ⇒
        tyTree_FAType (fun B : Type ⇒ tyTree_imp (tyTree_M A) (
        tyTree_imp (tyTree_imp (tyTree_base A) (tyTree_M B)) (
        tyTree_M B)))).
362 Definition bindt : to_ty bindTyTree := ·bind.
363
364 Definition l_bind (m : MTele): m:{T : TyTree & to_ty T} := ltac:(
        mrun (lift' bindt m)).
365
366 Definition li_bind : m:{T : TyTree & to_ty T} := ltac:(mrun (lift'
         bindt m)).
367
368 (* Check the result of li_bind *)
369 Eval cbn in mprojT1 (li_bind).
370 Eval cbn in to_ty (mprojT1 li_bind).
371 Eval cbn in mprojT2 li_bind.
372
373 (* Check the result of l_bind *)
374 Eval cbn in to_ty (mprojT1 (l_bind m)).
375 Eval cbn in mprojT2 (l_bind m).
376 Eval cbn in mprojT1 (l_bind m).
```

Listing A.1: Definición de LIFT y funciones auxiliares