Traducción del capítulo 12 de http://www.labri.fr/perso/casteran/CoqArt/coqartF.pdf[1]

Capítulo 12

* Caso de estudio

Los principios de la extracción de programas fueron presentados en el capítulo 11. Nosotros proponemos aquí un caso de estudio simple para ilustrar las sutilezas del vínculo entre Prop y Set. Veremos cómo el conocimiento del mecanismo de extracción permite facilitar la construcción de programas certificados sin descuidar las consideraciones de eficiencia.

La noción de árbol binario de búsqueda sirve de soporte para nuestro estudio. Nos proponemos construir programas certificados a través de la búsqueda, la inserción y la destrucción de información en esos árboles. El desarrollo completo se encuentra en las contribuciones de usuarios del sistema *Coq*[2]; aquí solo damos los detalles relativos a la extracción de programas.

12.1 Los árboles binarios de búsqueda

Clásicamente, un árbol binario de búsqueda es un árbol binario cuyas hojas no contienen ninguna información y cuyos nodos internos están etiquetados — en nuestro caso por enteros (de tipo Z)—; además se requiere que para todo nodo interno etiquetado como *n*, su subárbol izquierdo (respectivamente: derecho) solo contenga etiquetas estrictamente inferiores a *n* (respectivamente : estrictamente superiores). La figura 12.1 presenta uno de estos árboles.

12.1.1 Los árboles de búsqueda en *Coq*

En este desarrollo, no asociamos directamente un tipo *Coq* a los árboles binarios de búsqueda. Consideramos en primer lugar un tipo de datos — aquel de los árboles binarios etiquetados con enteros — y luego definimos el predicado ≪ ser de búsqueda ≫ sobre ese tipo. La definición de dicho predicado requiere algunas definiciones auxiliares.

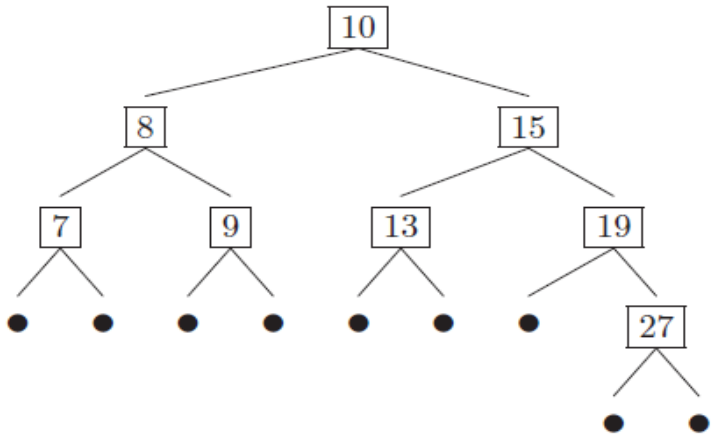

Figura 1: Un árbol binario de búsqueda

Estructura de datos

La sección 7.3.4 contiene una definición inductiva del tipo de los árboles binarios cuyos nodos internos son etiquetados con enteros; recordemos esta definición:

1 N. del T.: Marco así las partes de la traducción de las que no estoy segura o las partes que no supe traducir.

2 En el sitio http://coq.inria.fr/contribs-eng.html, cliquear sobre search-trees

```
Open Scope Z_scope.

Inductive Z_btree : Set :=
      Z_leaf : Z_btree
      | Z_bnode : Z→Z_btree→Z_btree→Z_btree.
```

A modo ilustrativo, damos el término *Gallina* que define el árbol de la Figura 1:

```
Z_bnode 10
      (Z_bnode 8
            (Z_bnode 7 Z_leaf Z_leaf)
            (Z_bnode 9 Z_leaf Z_leaf))
      (Z_bnode 15
            (Z_bnode 13 Z_leaf Z_leaf)
            (Z_bnode 19 Z_leaf (Z_bnode 27 Z_leaf Z_leaf)))
```

Noción de ocurrencia

Aquí damos una definición inductiva de la proposición "`occ n t`": ≪ el entero ***n*** tiene al menos una ocurrencia en el árbol *t* ≫ :

```
Inductive occ (n:Z) : Z_btree→Prop :=
| occ_root : ∀ t1 t2:Z_btree, occ n (Z_bnode n t1 t2)
| occ_l :
      ∀ (p:Z)(t1 t2:Z_btree), occ n t1 → occ n (Z_bnode p t1 t2)
| occ_r :
      ∀ (p:Z)(t1 t2:Z_btree), occ n t2 → occ n (Z_bnode p t1 t2).
```

Decidibilidad del test de ocurrencia : un enfoque naïve

Queremos desarrollar un programa certificado que permita testear si un entero *n* ocurre o no en un árbol *t*. En resumen, deseamos construir un término de *Gallina* que tenga el siguiente tipo:

```
      ∀ (n:Z)(t:Z_btree), {occ n t}+{~occ n t}
```

Una estrategia simple consiste en utilizar una recurrencia sobre *t*, así como la decidibilidad de la igualdad sobre Z (`lemma Z_eq_dec`). He aquí dicha construcción[3]:

```
Definition naive_occ_dec :
            ∀ (n:Z)(t:Z_btree), {occ n t}+{~occ n t}.
      induction t.
      right; auto with searchtrees.
      case (Z_eq_dec n z).
      induction 1; left; auto with searchtrees.
      case IHt1; case IHt2; intros; auto with searchtrees.
      right; intro H; elim (occ_inv H); auto with searchtrees.
      tauto.
Defined.
```

Usando el comando "`Extraction naive_occ_dec`", podemos observar el código extraído correspondiente a esa función:

```
let rec naive_occ_dec n = function
  Z_leaf ->Right
| Z_bnode (z1, z0, z)→
      (match Z_eq_dec n z1 with
        Left ->Left
      | Right →
                  (match naive_occ_dec n z0 with
```

3 La base searchtrees, propia de este desarrollo, contiene algunos lemas técnicos destinados a facilitar la automatización de pruebas; aquí no se detalla el contenido.

```
      Left ->Left
    | Right ->naive_occ_dec n z))
```

Notamos inmediatamente la falta de eficiencia de este programa: en el caso donde el entero *n* no ocurra en el árbol *t*, el diagnóstico (reenvío del valor `Right`, asimilado a `false`) es obtenido luego de una recorrida completa del árbol.

También podemos notar que la especificación de `naive_occ_dec` prohíbe cualquier mejora significativa: el segundo argumento de esta función está especificado como un árbol binario cualquiera, y nada le permite evitar una recorrida completa de este árbol en caso de ausencia de la información buscada. Esta falta de eficiencia puede ser evitada si se restringe el test de ocurrencia a una clase de árboles binarios que posean propiedades que hagan innecesaria una recorrida completa.

Caracterización de los árboles de búsqueda

Podemos definir de manera inductiva el predicado ≪ ser un árbol binario de búsqueda ≫ :

— Toda hoja es un árbol binario de búsqueda.

— Si t_1 y t_2 son árboles binarios de búsqueda, y *n* es estrictamente mayor que toda etiqueta de t_1 y menor que toda etiqueta de t_2, entonces el árbol de raíz *n* , de hijo izquierdo t_1 e hijo derecho t_2 es un árbol binario de búsqueda.

La formalización en *Coq* se hace en tres etapas:

1. Definición de un predicado à deux places "`min z t`" : ≪ *z* es menor que toda etiqueta de *t* ≫,

2. ídem para "`maj z t`" : ≪ *z* es mayor que toda etiqueta de *t* ≫,

3. Definición inductiva de `search_tree: Z_btree → Prop`, utilizando `maj` y `min` como predicados auxiliares.

```
Inductive min (n:Z)(t:Z_btree) : Prop :=
      min_intro : (∀ p:Z, occ p t → n < p)→ min n t.

Inductive maj (n:Z)(t:Z_btree) : Prop :=
      maj_intro : (∀ p:Z, occ p t → p < n)→ maj n t.

Inductive search_tree : Z_btree→Prop :=
      | leaf_search_tree : search_tree Z_leaf
      | bnode_search_tree :
        ∀ (n:Z)(t1 t2:Z_btree),
        search_tree t1 → search_tree t2 → maj n t1 → min n t2 →
        search_tree (Z_bnode n t1 t2).
```

Nota 12.1 Puede parecer sorprendente que `min` y `maj` estén definidos de manera inductiva; el lector puede encontrar más natural una simple definición de la siguiente forma:

```
Definition min (n:Z)(t:Z_btree) : Prop :=
      ∀ p:Z, occ p t → n < p.
```

El autor de este texto prefirió la comodidad de un tipo inductivo frente a un solo constructor, que le permite utilizar las tácticas `split` en introducción y `case` en eliminación, mientras que la definición anterior obligaría a controlar la δ-expansión de `min` para la técnica `unfold`; por otra parte, una utilización mal controlada de `unfold` podría provocar expansiones de `min` no deseables, y perturbar la legibilidad de los objetivos. Con la solución elegida, las construcciones y los análisis por caso de proposiciones de la forma " `min n t`" solo se efectúan en caso de necesidad. Este es un método general que aconsejamos utilizar con frecuencia.

Esta elección entre una definición simple y un tipo inductivo de un solo constructor se encuentra también en programación; consideremos por ejemplo en OCAML la definición de un tipo para definir las variables indizadas por enteros (por ejemplo en un compilador). Nosotros

preferimos la definición de un nuevo tipo:

```
type variable = Mkvar of int
```

frente a un simple sinónimo:

```
type variable = int
```

Ejercicio 12.1 ** Los predicados `min` y `maj` podrían haber sido escritos directamente con una definición inductiva recursiva sobre el tipo `Z_btree` (sin utilizar explícitamente el predicado `occ`). Retomar el desarrollo sobre los árboles binarios de búsqueda con esta nueva definición. Deberá comparar la comodidad de la utilización respectiva de los dos enfoques. Se debe procurar minimizar el trabajo de modificación debido al cambio pedido. En este sentido, el mantenimiento de pruebas presenta los mismos problemas y soluciones que en la ingeniería de software.

12.2 Especificación de programas

Las especificaciones de programas de búsqueda, de agregado y de eliminación en los árboles binarios de búsqueda están dados bajo la forma de tipos como `Set`; estas especificaciones utilizan los predicados `occ` y `search_tree`, así como las construcciones `sig` et `sumbool`, presentadas en las secciones 10.1.1 y 10.1.3.

12.2.1 Test de ocurrencia

Una función de búsqueda de un entero *p* en un árbol *t* debe calcular un valor precisando si *p* tiene o no una ocurrencia en *t*. El tipo `sumbool` nos permite exprimir la relación entre el valor calculado (`left` o `right`) y la certeza de que *n* aparece o no en *t*. Además, la construcción de un programa eficiente de test de ocurrencia puede (¿debe?) presuponer que *t* es un árbol de búsqueda.

Proponemos entonces la especificación siguiente para el valor devuelto, bajo la forma de un tipo paramétrico por *p* y *t*:

```
Definition occ_dec_spec (p:Z)(t:Z_btree) : Set :=
      search_tree t → {occ p t}+{~occ p t}.
```

La especificación del programa a construir es por tanto la siguiente:

```
      ∀ (p:Z)(t:Z_btree), occ_dec_spec p t
```

12.2.2 Programa de inserción

La especificación de una función para insertar una ocurrencia de *p* en un árbol *t* debe precisar una relación entre las ocurrencias de enteros en t y el árbol resultante de dicha inserción. Por otra parte, esta especificación debe permitir rechazar cualquier ejecución que no produzca un árbol de búsqueda. En caso contrario, una solución trivial pero que no interesa, consistente en la construcción del árbol "`Z_bnode` *n t* `Z_leaf`" satisfaría la especificación.

Predicado asociado a la inserción

Por las mismas razones que en `min` y `maj`, utilizamos un predicado inductivo para formalizar la relación ≪ *t′* se obtiene por la inserción de *n* en *t*>>, que denotaremos en *Coq* "INSERT *n t t′* ". Este predicado expresa las siguientes propiedades:

— Todo entero que aparezca en *t* debe aparecer en t'.

— *t′* contiene al menos una ocurrencia de *n*

— Todo entero que aparezca en *t'* aparece igualmente en *t′*, o es igual a *n*, que aparece en *t* [4]

4 N. del T.: Creo que hay un error de tipeo, el texto original dice "Tout entier apparaissant dans t' apparait egalement dans t′, ou est egal a n, apparaissant dans t,"

— t′ es un árbol de búsqueda.

```
Inductive INSERT (n:Z)(t t′:Z_btree) : Prop :=
      insert_intro :
      (∀ p:Z, occ p t → occ p t′)→ occ n t′ →
      (∀ p:Z, occ p t′ → occ p t ∨ n = p)→ search_tree t′ →
      INSERT n t t′.
```

Especificación de la función de inserción

El programa certificado de inserción debe asociar a todo entero *n* y a todo árbol de búsqueda *t* un árbol *t′*, acompañado de una prueba de la proposición "`INSERT` *n t t′* ". Podemos entonces construir la especificación (dependiente) siguiente:

```
Definition insert_spec (n:Z)(t:Z_btree) : Set :=
      search_tree t → {t′:Z_btree | INSERT n t t′}.
```

La especificación del programa a construir es entonces la siguiente:

```
      ∀ (n:Z)(t:Z_btree), insert_spec n t
```

12.2.3 Programa de eliminación

Para la eliminación de una ocurrencia en un árbol, lo que hay que hacer es muy similar al caso de la inserción: damos sin más comentarios la definición de un predicado RM y del tipo dependiente rm_spec:

```
Inductive RM (n:Z)(t t′:Z_btree) : Prop :=
      rm_intro :
      ~occ n t′ →
      (∀ p:Z, occ p t′ → occ p t)→
      (∀ p:Z, occ p t → occ p t′ ∨ n = p)→
      search_tree t′ →
      RM n t t′.

Definition rm_spec (n:Z)(t:Z_btree) : Set :=
      search_tree t → {t′ : Z_btree | RM n t t′}.
```

El tipo del programa certificado a construir será entonces el siguiente:

```
      ∀ (n:Z)(t:Z_btree), rm_spec n t
```

Ejercicio 12.2 ** ¿Cómo quedan las especificaciones anteriores si elegimos definir un tipo ≪ árbol binario de búsqueda ≫ de tipo Set ?

12.3 Lemas preliminares

Una vez especificados los programas a construir, sería absurdo comenzar su desarrollo sin trabajo previo. El usuario que se lanzara en una misión así, sería rápidamente bloqueado por la acumulación de objetivos a resolver.

A modo de ejemplo, he aquí uno de los muchos objetivos secundarios que pueden aparecer durante el desarrollo de programas certificados que operan sobre árboles de búsqueda (este ejemplo se extrae de la elaboración del programa de destrucción).

```
...
n : Z
p : Z
t1 : Z_btree
t2 : Z_btree
```

t' : Z_btree
H : n < p
H0 : search_tree (Z_bnode n t1 t2)
H1 : RM p t2 t'
H2 : occ p (Z_bnode n t1 t')

D : occ p t1 ∨ *occ p t'*

==========

~*occ p t1*

Como esta situación se presenta varias veces en nuestro desarrollo, parece útil construir una mini-biblioteca de lemas técnicos sobre los árboles de búsqueda, que expresan las propiedades simples: la situación descrita anteriormente se resuelve (a veces de forma automática) si se ha probado de antemano el siguiente lema:

```
Lemma not_left :
  ∀ (n:Z)(t1 t2:Z_btree),
      search_tree (Z_bnode n t1 t2)→∀p:Z, p ≥ n → ~occ p t1.
```

Para mostrar qué tipo de resultados han servido para facilitar el resto del desarrollo, damos sólo las declaraciones de todos estos lemas técnicos en la Figura 2.

12.4 Hacia la realización/¿implementación?

Una vez preparado el terreno, no queda más que desarrollar los programas certificados; a diferencia de la primera tentativa presentada en 12.1.1, no queremos correr nuevamente el riesgo de tener programas ineficientes al guiarnos demasiado por la interacción con Coq y las herramientas de automatización. Dentro de las herramientas actualmente disponibles para guiar la construcción de un programa certificado a partir de una intuición computacional, utilizamos en este caso la táctica `refine` ya presenta en la sección 10.2.7.

12.4.1 Realización/implementación del test de ocurrencia

Recordemos que el objetivo es proporcionar un término para la especificación siguiente:

```
Definition occ_dec : ∀ (p:Z)(t:Z_btree), occ_dec_spec p t.
```

Podemos evacuar el caso más simple, donde el árbol se reduce a una hoja; sabemos que el entero *p* no puede aparecer en Z_leaf, y por tanto el constructor `right` del tipo `sumbool` es apropiado.

En el caso general de un árbol de la forma " `Z_bnode` *n t1 t2* ", debemos comparar *n* y *p* para buscar la ocurrencia de p, ya sea en la raíz *n*, en *t1* o en *t2*. La decidibilidad del orden total en `Z` se expresa en de *Coq* utilizando los dos teoremas siguientes, tomados de la biblioteca `ZArith`:

```
Z_le_gt_dec : ∀ x y:Z, {x ≤ y}+{x > y}
Z_le_lt_eq_dec : ∀ x y:Z, x ≤ y → {x < y}+{x = y}
```

```
Lemma min_leaf : ∀ z:Z, min z Z_leaf.

Lemma maj_leaf : ∀ z:Z, maj z Z_leaf.

Lemma maj_not_occ : ∀ (z:Z)(t:Z_btree), maj z t → ~occ z t.

Lemma min_not_occ : ∀ (z:Z)(t:Z_btree), min z t → ~occ z t.

Section search_tree_basic_properties.
      Variable n : Z.
      Variables t1 t2 : Z_btree.
      Hypothesis se : search_tree (Z_bnode n t1 t2).

      Lemma search_tree_l : search_tree t1.

      Lemma search_tree_r : search_tree t2.

      Lemma maj_l : maj n t1.

      Lemma min_r : min n t2.

      Lemma not_right : ∀ p:Z, p ≤ n → ~occ p t2.

      Lemma not_left : ∀ p:Z, p ≥ n → ~occ p t1.

      Lemma go_left :
            ∀ p:Z, occ p (Z_bnode n t1 t2)→ p < n → occ p t1.

      Lemma go_right :
            ∀ p:Z, occ p (Z_bnode n t1 t2)→ p > n → occ p t2.

End search_tree_basic_properties.

Hint Resolve go_left go_right not_left not_right
      search_tree_l search_tree_r maj_l min_r : searchtrees.
```

Figura 2: Lemas técnicos sobre árboles de búsqueda

Estos dos teoremas, aplicados a *n* y *p*, permiten distinguir los tres casos siguientes:

- si $p < n$, el resultado de una invocación recursiva " `occ_dec p t1` " determina directamente el resultado de la invocación principal:
  - si " `occ_dec` *p t1* " se reduce a " `(left _ _` π", donde π es una prueba de " `occ` *p t1* ", entonces retorna " `left _ _` π' ", donde π' es un término de prueba de " `occ p (Z_bnode` *n t1 t2*`)` ",
  - si " `occ_dec` *p t1* " se reduce a " `right _ _` π", donde π es una prueba de " ~`occ` *p t1* ", entonces retorna " `right _ _` π' ", donde π′ es un término de prueba de " ~ `occ` *p* (`Z_bnode` *n t1 t2*) ". La construcción de π′ a partir de π es una aplicación de `go_left`.
- si $p = n$, retorna " `left _ _` π", donde π es una prueba de la proposición " `occ` *n* (`Z_bnode` *n t1 t2*) "
- si $p > n$, se aplica un enfoque simétrico al del caso $p<n$.

Utilizando `refine`, le damos a *Coq* un término cuyos componentes lógicos (que, recordemos, nos son indiferentes) se sustituyen por el símbolo '_'. La implementación de `occ_dec` muestra que esas indeterminaciones son en su mayoría levantadas por `refine` o por los lemas

técnicos colocados en la base `searchtrees`:

```
Definition occ_dec : ∀ (p:Z)(t:Z_btree), occ_dec_spec p t.
  refine
    (fix occ_dec (p:Z)(t:Z_btree){struct t} : occ_dec_spec p t :=
      match t as x return occ_dec_spec p x with
      | Z_leaf ⇒ fun h ⇒ right _ _
      | Z_bnode n t1 t2 ⇒
        fun h ⇒
          match Z_le_gt_dec p n with
          | left h1 ⇒
            match Z_le_lt_eq_dec p n h1 with
            | left h'1 ⇒
              match occ_dec p t1 _ with
              | left h''1 ⇒ left _ _
              | right h''2 ⇒ right _ _
              end
            | right h'2 ⇒ left _ _
            end
          | right h2 ⇒
            match occ_dec p t2 _ with
            | left h''1 ⇒ left _ _
            | right h''2 ⇒ right _ _
            end
          end
    end); eauto with searchtrees.
  rewrite h'2; auto with searchtrees.
Defined.
```

El comando " `Extraction occ_dec` " nos permite recuperar el contenido algorítmico de nuestro programa en sintaxis OCAML. Nótese que los constructores `Left` et `Right` deben asimilarse respectivamente a `true` y `false`.

```
let rec occ_dec p = function
| Z_leaf -> Right
| Z_bnode (n, t1, t2) ->
  (match z_le_gt_dec p n with
   | Left ->
     (match z_le_lt_eq_dec p n with
        Left -> occ_dec p t1
      | Right -> Left)
   | Right -> occ_dec p t2)
```

El programa obtenido mediante extracción realiza un recorrido de una sola rama del árbol binario de búsqueda, guiado por las comparaciones entre el número buscado y las raíces encontradas sucesivamente. Con la notación más convencional, habríamos obtenido el siguiente programa OCAML:

```
let rec occ_dec p t =
  match t with
    Z_leaf -> false
  | Z_bnode(n,t1,t2) ->
      if p <= n
      then if p < n then occ_dec p t1 else true
      else occ_dec p t2
```

12.4.2 Inserción

El enfoque para la inserción de un entero en un árbol de búsqueda es similar a la del test de

ocurrencia; así que mostraremos solamente las diferencias destacables respecto al estudio precedente.

Análisis a la Prolog

La programación de la inserción en un árbol de búsqueda es más compleja que la del test de ocurrencia; de hecho se trata de construir un nuevo árbol, de asegurarse que efectivamente es un árbol de búsqueda, mientras que la función `occ_dec` solo retorna un booleano. Para paliar esta dificultad adicional, hemos elegido comenzar el desarrollo del programa certificado de inserción con la prueba de una suite de lemas sobre el predicado `INSERT`, que es muy similar al paquete de cláusulas que escribiría un programador Prolog para definir este predicado. Damos en la Figura 3 estos lemas principales (sin sus pruebas).

Construcción con `refine`

La colocación de estos cuatro lemas en la base `searchtrees` facilita notablemente la construcción con `refine` del programa certificado `insert`, presentado en la Figura 4.

Tengamos en cuenta que los tipos de los tres argumentos de la función `insert` pertenecen a los tipos `Set` y `Prop`. El resultado retornado por `insert` es un par donde una componente es de tipo `Set` y la otra de tipo `Prop`. La distinción entre estos tipos juega un rol importante para controlar la cantidad de cálculos que son efectuados en la función extraída. Dentro de la función *Coq*, se incluyen algunos cálculos para construir el componente de prueba del resultado. Al momento de la extracción, el argumento de la función `insert` que es una prueba desaparece, así como el componente prueba del resultado y los cálculos efectuados para construirlo. Incluso si las funciones fuertemente especificadas parecen contener más cálculos que las funciones débilmente especificadas, sus correspondientes extractos pueden ser igualmente eficaces si el diseñador se encargó de colocar solo los datos pertinentes en el tipo Set.

¿Hace falta un test de decisión para `search_tree` ?

Ya hemos resaltado que muchos de nuestros lemas y programas certificados utilizan una precondición de la forma “ `search_tree t` ”; donde el predicado `search_tree` tiene tipo `Z_btree→Prop` y no debe ser confundido con una función booleana que pueda ser utilizada en los programas. Se podría considerar el desarrollo de un test de decisión para `search_tree`, que tuviera el siguiente tipo:

```
search_tree_dec : ∀ t:Z_btree, {search_tree t}+{~search_tree t}
```

Nosotros no elegimos este método, aplicable a cualquier instancia de Z_btree. Es más natural considerar que los árboles manipulados por nuestros programas se construyen a partir de árboles vacíos con inserciones sucesivas.

Por ejemplo, especificamos la construcción de un árbol de búsqueda a partir de una lista de números enteros, de modo que el árbol obtenido contiene exactamente los elementos de la lista pasada como argumento:

```
Definition list2tree_spec (l:list Z) : Set :=
  {t : Z_btree | search_tree t ∧ (∀ p:Z, In p l ↔ occ p t)}.
```

Lo interesante de tal especificación es que permite la construcción de árboles de búsqueda complejos, sin tener que comprobar en cada paso que la inserción de un entero sólo se produce en un árbol de búsqueda.

Para el desarrollo de este programa de conversión de listas en árboles búsqueda, tomamos un enfoque clásico programación funcional, que consiste en una llamada a una función recursiva auxiliar.

Comencemos especificando esta función, que toma como argumento una lista *l* y un árbol de búsqueda *t* para construir un árbol de búsqueda *t'* que contiene exactamente la unión de los elementos de *l* y *t*:

```
Definition list2tree_aux_spec (l:list Z)(t:Z_btree) :=
  search_tree t →
    {t' : Z_btree | search_tree t' ∧ (∀ p:Z, In p l ∨ occ p t ↔ occ p t')}.
```

Sólo falta proponer - siempre a través de `refine` - una implementación para `list2tree_aux_spec` y `list2tree_spec`. El término provisto como argumento para `refine` en la implementación de la función auxiliar puede parecer un poco complejo, invitamos a los lectores a estudiarlo de cerca.

```
Definition list2tree_aux :
  ∀ (l:list Z)(t:Z_btree), list2tree_aux_spec l t.
refine
  (fix list2tree_aux (l:list Z) :
    ∀ t:Z_btree, list2tree_aux_spec l t :=
    fun t ⇒
      match l return list2tree_aux_spec l t with
      | nil ⇒ fun s ⇒ exist _ t _
      | cons p l' ⇒
        fun s ⇒
          match insert p (t:=t) s with
          | exist t' _ ⇒
            match list2tree_aux l' t' _ with
            | exist t'' _ ⇒ exist _ t'' _
            end
          end
      end).
...
Defined.

Definition list2tree : ∀ l:list Z, list2tree_spec l.
 refine
  (fun l ⇒ match list2tree_aux l (t:=Z_leaf) _ with
                    | exist t _ ⇒ exist _ t _
                    end).
...
Defined.
```

Programas extraídos

Los programas extraídos de las funciones `insert` y `list2tree` son muy simples; Catherine Parent[70] y Jean-Christophe Filliatre[40] y Antonia Balaa [6] estudiaron cómo hacer estas construcciones de programas certificados menos detalladas. Podemos esperar que las futuras versiones de de *Coq* permitan aumentar la simplicidad de las descripciones de algoritmos.

```
let rec insert n = function
| Z_leaf -> Z_bnode (n, Z_leaf, Z_leaf)
| Z_bnode (p, t1, t2) ->
  (match z_le_gt_dec n p with
   | Left ->
     (match z_le_lt_eq_dec n p with
      | Left -> Z_bnode (p, (insert n t1), t2)
      | Right -> Z_bnode (n, t1, t2))
   | Right -> Z_bnode (p, t1, (insert n t2)))

let rec list2tree_aux l t =
  match l with
```

```
    | Nil -> t
    | Cons (p, l') -> list2tree_aux l' (insert p t)

let list2tree l =
  list2tree_aux l Z_leaf
```

12.4.3 ** Destrucción

La destrucción de un elemento en un árbol binario de búsqueda no supone más problemas que la inserción, excepto en el caso en que la cumbre a destruir es la raíz del árbol considerado; la solución clásica para la eliminación de *n* de un árbol de búsqueda "`Z_bnode` *n t2 t1*" es eliminar de *t1* su mayor elemento *q* y devolver el árbol `Z_bnode` *q r t2*" donde *r* es el árbol obtenido por eliminación de *q* en *t1*. En el caso de que t1 se reduce a una hoja, alcanza con devolver el árbol t2.

Desde el punto de vista de la programación, vemos que es necesario especificar e implementar la operación auxiliar "eliminar la mayor etiqueta de un árbol de búsqueda no vacío", para utilizarla luego en la función de destrucción principal.

Procedemos como en los apartados anteriores, definiendo un predicado inductivo RMAX:

```
Inductive RMAX (t t':Z_btree)(n:Z) : Prop :=
  rmax_intro :
  occ n t →
  (∀ p:Z, occ p t → p ≤ n)→
  (∀ q:Z, occ q t' → occ q t)→
  (∀ q:Z, occ q t → occ q t' ∨ n = q)→
  ~occ n t' → search_tree t' → RMAX t t' n.
```

Después de probar una serie de "lemas a la Prolog" sobre RMAX, construimos una función para eliminar el mayor entero de un árbol búsqueda no vacío; notar el uso de un tipo inductivo sigS (ver sección 10.1.2).

```
Definition rmax_sig (t:Z_btree)(q:Z) :=
  {t':Z_btree | RMAX t t' q}.

Definition rmax_spec (t:Z_btree) :=
  search_tree t → is_bnode t → {q:Z & rmax_sig t q}.

Definition rmax : ∀ t:Z_btree, rmax_spec t.
...
```

La implementación de la función de supresión en un árbol de búsqueda continúa como en el caso de la inserción. No damos los detalles de esta parte, el lector los podrá encontrar en las contribuciones de *Coq*. Es interesante, sin embargo, terminar con el código obtenido mediante extracción de `rm` y `Rmax`.

```
let rec rmax = function
| Z_leaf -> assert false (* absurd case *)
| Z_bnode (r, t1, t2) ->
  (match t2 with
   | Z_leaf -> ExistS (r, t1)
   | Z_bnode (n', t'1, t'2) ->
     let ExistS (num, r0) = rmax t2 in
       ExistS (num, (Z_bnode (r, t1, r0))))

let rec rm n = function
| Z_leaf -> Z_leaf
| Z_bnode (p, t1, t2) ->
  (match z_le_gt_dec n p with
```

```
| Left ->
  (match z_le_lt_eq_dec n p with
   | Left -> Z_bnode (p, (rm n t1), t2)
   | Right ->
     (match t1 with
      | Z_leaf -> t2
      | Z_bnode (p', t'1, t'2) ->
        let ExistS (q, r) = rmax (Z_bnode (p', t'1, t'2)) in
          Z_bnode (q, r, t2)))
| Right -> Z_bnode (p, t1, (rm n t2)))
```

Tenga en cuenta el término "`assert false`" en el código extraído de `Rmax`, que corresponde a la precondición "`is_bnode t`", en conflicto con la estructura recursiva primitiva de la función, que debe por tanto examinar el caso `t = Z_leaf`.

12.5 Mejoras posibles

Los árboles de búsqueda presentados anteriormente solo permiten representar conjuntos finitos de enteros. Pero las únicas propiedades Z que hemos utilizado son las de la relación de orden ≤, sobre todo el hecho de que la comparación de dos números enteros es decidible (teoremas `Z_le_lt_eq_dec` y `Z_le_gt_dec`). Debemos poder generalizar nuestro enfoque a cualquier tipo que posea estas características: no sólo `nat`, sino también `nat*nat`, `list bool`, etc..

Por otra parte, una utilización frecuente de los árboles de búsqueda es la representación de funciones de dominio finito, lo que no es permitido por nuestra aplicación juguete.

El siguiente capítulo trata sobre la manera de superar estas deficiencias, con el nuevo sistema de módulos de *Coq*. Un desarrollo único permite la representación de funciones de dominio finito; los árboles de búsqueda se utilizan siempre que el dominio tenga una relación de orden total donde la comparación es decidible.

12.6 Otro ejemplo

El capítulo 12 del libro de Jean-Francois Monin [67] ilustra las posibilidades de Coq sobre la especificación y la derivación de un programa de búsqueda en tablas. Inicialmente, el problema está especificado de forma muy concisa y en términos muy generales a través de ≪ types riches ≫. Usando una sencilla aplicación, puede cómodamente deducir a partir de allí versiones especializadas en tablas o listas, que se presten a la elaboración de una solución algorítmica obtenida por extracción.

```
Lemma insert_leaf :
  ∀ n:Z, INSERT n Z_leaf (Z_bnode n Z_leaf Z_leaf).

Lemma insert_l :
  ∀ (n p:Z)(t1 t'1 t2:Z_btree),
      n < p → search_tree (Z_bnode p t1 t2)→ INSERT n t1 t'1 →
      INSERT n (Z_bnode p t1 t2)(Z_bnode p t'1 t2).

Lemma insert_r :
  ∀ (n p:Z)(t1 t2 t'2:Z_btree),
      n > p → search_tree (Z_bnode p t1 t2)→ INSERT n t2 t'2 →
      INSERT n (Z_bnode p t1 t2)(Z_bnode p t1 t'2).

Lemma insert_eq :
  ∀ (n:Z)(t1 t2:Z_btree), search_tree (Z_bnode n t1 t2)→
      INSERT n (Z_bnode n t1 t2)(Z_bnode n t1 t2).

Hints Resolve insert_leaf insert_l insert_r insert_eq
      : searchtrees.
```

Figura 3: Lemas a la Prolog para la inserción

```
Definition insert : ∀ (n:Z)(t:Z_btree), insert_spec n t.
    refine
      (fix insert (n:Z)(t:Z_btree){struct t} : insert_spec n t :=
      match t return insert_spec n t with
      | Z_leaf ⇒ fun s ⇒ exist _ (Z_bnode n Z_leaf Z_leaf) _
      | Z_bnode p t1 t2 ⇒
        fun s ⇒
            match Z_le_gt_dec n p with
            | left h ⇒
                  match Z_le_lt_eq_dec n p h with
                  | left _ ⇒
                        match insert n t1 _ with
                        | exist t3 _ ⇒ exist _ (Z_bnode p t3 t2) _
                        end
                  | right h' ⇒ exist _ (Z_bnode n t1 t2) _
                  end
            | right _ ⇒
            match insert n t2 _ with
            | exist t3 _ ⇒ exist _ (Z_bnode p t1 t3) _
            end
      end
      end); eauto with searchtrees.
    rewrite h'; eauto with searchtrees.
Defined.
```

Figura 4: Desarrollo de un programa de inserción