Implementación de Árboles Patricia en Coq

Víctor Zamora Gutiérrez

9 de febrero de 2021

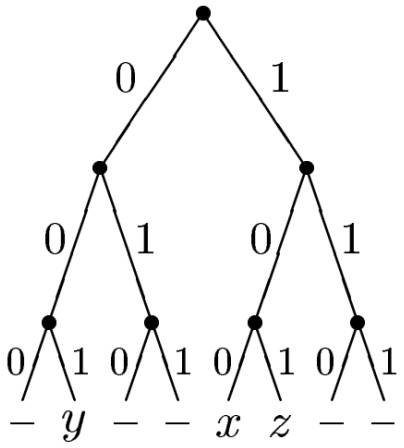
Resumen

La meta de este trabajo fue realizar una implementación de árboles Patricia en Coq. Para esta implementación, era deseable demostrar que las operaciones principales funcionaban; con esto nos referimos a las operaciones de inserción y unión de árboles. La meta se logró de manera satisfactoria para la operación de inserción. Para la de unión, hubieron algunos problemas que evitaron que demostráramos propiedades sobre ella, pero al menos se demostró que la función utiliza recursión bien fundada.

1. Introducción

Los Árboles Patricia[] son estructuras de datos que funcionan como diccionarios con llaves enteras. Son atractivos debido a la sencillez con la que se implementan en lenguajes funcionales y debido a su eficiencia. La idea básica es que se tiene un árbol en el que el 0 representa "izquierda" y el 1 representa "derecha". Por ejemplo, en el árbol de la figura 1, el elemento y tiene llave 4, pues $4 = 100$ y para llegar a y tenemos que tomar el camino izquierda-izquierda-derecha (leyendo el número de adelante hacia atrás). Como podemos notar, los elementos siempre estarán en hojas de los árboles.

Figura 1: Figura obtenida en []



Por supuesto, si los árboles Patricia estuvieran implementados como el de la figura 1, no serían muy útiles, pues las búsquedas tomarían tiempo lineal sobre el tamaño de las llaves, lo cual no es muy óptimo. Por esto, de esta idea básica se obtienen varias optimizaciones para crear una estructura de datos eficiente, de las cuales hablaremos más adelante.

En este trabajo se decidió hacer una implementación de árboles Patricia en Coq. Se comenzó desde lo más básico, que son los tries binarios (estructuras de la figura 1) hasta llegar a la estructura final. Para la estructura final, se implementaron las operaciones de búsqueda, inserción y unión, de las cuales también hablaremos más adelante. Además se verificó el funcionamiento de la inserción con respecto a búsquedas. Para la unión no se verificó nada debido a la forma en la que tuvo que definirse; sin embargo se logró definir a pesar de que utiliza recursión distinta a la que se trabajó con Coq a lo largo del semestre.

2. Estructura del Proyecto

El proyecto está dividido en dos tipos de archivos: de definiciones y de proposiciones. Los archivos de definiciones son:

-

3. Herramientas

Para el desarrollo del proyecto, utilizamos algunas bibliotecas auxiliares de Coq. La más importante de estas fue la biblioteca `BinNat`[] que define a los números naturales en binario. Esta biblioteca se usó para definir las llaves y la estructura general de los árboles. Además de esta, utilizamos las siguientes bibliotecas:

- `BinPos` - Para enteros positivos binarios (naturales sin cero).
- `Lia` - Para demostrar propiedades de operaciones aritméticas.
- `Bool.Bool` - Para propiedades sobre el tipo `Bool` de Coq.

4. Desarrollo

4.1. Tries binarios

Decidimos basarnos completamente en[] para el desarrollo del proyecto. Esto implicó una optimización gradual del trie binario hasta llegar a árboles Patricia.

El trie binario es la estructura de datos de la figura 1. Básicamente es un árbol binario en el que cada arista está etiquetada con 0 o 1, dependiendo de si es arista izquierda o derecha. Los nodos internos no contienen ninguna información; son las hojas quienes contienen a los elementos del árbol. Nuestra definición del trie binario se encuentra en el listado 1.

```
(* Empezamos por definir tries binarios como en el artículo. *)
Inductive binTrie1: Type :=
| empty1 : binTrie1
| leaf1: nat -> binTrie1
| trie1: binTrie1 -> binTrie1 -> binTrie1.
```

Listado 1: Definición de trie binario

La búsqueda en trie binario es sencilla. Para buscar una llave x simplemente vemos si es par o impar. Si es par, buscamos por el lado izquierdo y si es impar buscamos por el derecho. Dividimos la llave entre dos y continuamos el proceso hasta llegar a una hoja. La función de búsqueda está en el listado 2.

```
(* Función de búsqueda *)
Fixpoint lookup1 (key: N) (t: binTrie1) : option nat :=
match t with
| empty1 => None
| leaf1 x => Some x
| trie1 t1 t2 => if (N.even key) then lookup1 (N.div2 key) t1
                 else lookup1 (N.div2 key) t2
end.
```

Listado 2: Búsqueda en un trie binario

4.2. Primera Optimización

La primera optimización que se hace para los tries binarios es colapsar a dos subárboles vacíos en un solo árbol. Esto para decrementar el tamaño del árbol. Dicha optimización se logra por medio de un *constructor inteligente*. La idea de los constructores inteligentes es que en lugar de utilizar el constructor `trie` del listado 1, utilizaremos únicamente el constructor inteligente para construir nuestros árboles. El constructor inteligente se enecuentra en el listado 3.

```
(* Constructor inteligente *)
Definition smartTrie1 (t1 t2: binTrie1) : binTrie1 :=
match t1, t2 with
| empty1, empty1 => empty1
| _, _ => trie1 t1 t2
end.
```

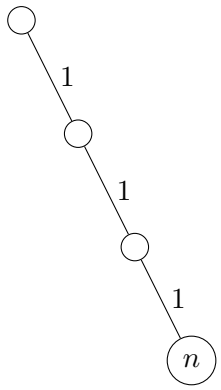
Listado 3: Primera optimización de los tries binarios

4.3. Segunda Optimización

La segunda optimización sale de la siguiente consideración: supongamos que tenemos un árbol cuyo único elemento es una hoja y el resto de sus subárboles son vacíos. En la figura 2 podemos ver un árbol así. Este árbol tiene un solo elemento; sin embargo, tenemos que recorrerlo todo para encontrar

este elemento. Esto no tiene mucho sentido, y de aquí surge la idea de colapsar este tipo de árboles en un solo nodo. Para lograr esto, se usó la optimización del listado 4. Este listado contiene otro constructor inteligente que, al encontrar una hoja junto a un árbol vacío, colapsa el árbol vacío con la hoja.

Figura 2: Árbol con una sola hoja




```
(* Nuevo constructor inteligente. *)
Definition smartTrie2 (t1 t2: binTrie2): binTrie2 :=
match t1, t2 with
| empty2, empty2 => empty2
| leaf2 j x, empty2 => leaf2 (2*j) x
| empty2, leaf2 j x => leaf2 (2*j+1) x
| _, _ => trie2 t1 t2
end.
```

Listado 4: Segunda optimización de los tries binarios

Para que esta optimización funcionara, tuvimos que modificar la estructura de árbol para que las hojas guardaran un prefijo de su llave. La nueva estructura de árbol está en el listado 5.

```
(* Trie binario en el que las hojas guardan parte de la llave *)
Inductive binTrie2: Type :=
| empty2: binTrie2
| leaf2: N -> nat -> binTrie2
| trie2: binTrie2 -> binTrie2 -> binTrie2.
```

Listado 5: Árboles cuyas hojas guardan una llave

Observemos que los prefijos que guardan las hojas son de tipo `N`. Este es el tipo para números binarios en Coq.

Después de realizar esta optimización, hay que modificar ligeramente la función de búsqueda para que al llegar a una hoja, revise que el prefijo de la hoja sea igual al elemento que busca.

4.4. Tercera optimización

Una alternativa a la optimización anterior es que en lugar de que las hojas guarden un prefijo de su llave, guarden la llave completa. Así, en la función de búsqueda ya no tenemos que descartar bits (o equivalentemente, dividir). Para que esto funcione, es necesario que los nodos guarden el bit sobre el que se dividen. Lo que esto quiere decir es que cada nodo guarda su profundidad, y así la función de búsqueda solo tiene que revisar el bit correspondiente a la profundidad del nodo.

Como guardar la profundidad complica los cálculos de buscar el bit correspondiente, en la práctica lo que se hace es guardar un entero binario con un solo bit prendido, y revisar dicho bit en la llave buscada.

Para esto, definimos la nueva estructura dada por el listado 6. La nueva función de búsqueda se encuentra en el listado 7. La función `zeroBit` nos dice si el bit prendido de `m` está apagado en `key`. Podemos verla en el listado 8.

```
(* Tipo de tries binarios *)
Inductive binTrie3: Type :=
| empty3 : binTrie3
| leaf3: N -> nat -> binTrie3
| trie3: N -> binTrie3 -> binTrie3 -> binTrie3.
```

Listado 6: Estructura de datos definida de acuerdo a la nueva optimización

```
(* Nueva función de búsqueda *)
Fixpoint lookup3 (key: N) (t: binTrie3) : option nat :=
match t with
| empty3 => None
| leaf3 j x => if (N.eqb j key) then Some x
               else None
| trie3 m t1 t2 => if (zeroBit key m) then lookup3 key t1
                  else lookup3 key t2
end.
```

Listado 7: Nueva función de búsqueda

```
(* Nos dice si el m-ésimo bit de k es 0. *)
Definition zeroBit (k m: N) : bool :=
N.eqb (N.land k m) 0.
```

Listado 8: Función que revisa si un bit está apagado