Implementación de Árboles Patricia en Coq

Víctor Zamora Gutiérrez

9 de febrero de 2021

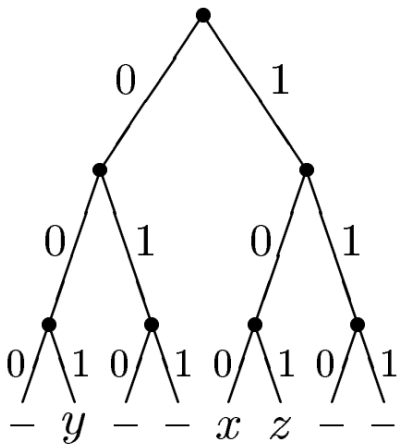
Resumen

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[1] 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 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 [1]



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:

- `Defs_Bin.v` - Contiene definiciones de funciones sobre números binarios.
- `Defs_Misc.v` - Contiene definiciones miscelaneas.
- `Defs_Patricia.v` - Contiene definiciones de árboles Patricia.
- `Defs_Trie.v` - Contiene definiciones de tries binarios y sus optimizaciones.

Los archivos de proposiciones son:

- `Props_Bin.v` - Contiene proposiciones de funciones sobre números binarios.
- `Props_Misc.v` - Contiene proposiciones miscelaneas.
- `Props_Patricia.v` - Contiene proposiciones de árboles Patricia.
- `Props_Trie.v` - Contiene proposiciones de tries binarios y sus optimizaciones.

Para que el proyecto funcione correctamente, es necesario compilar cada archivo por separado. Esto se puede hacer desde CoqIde con Compile → Compile buffer.

Se utilizó la versión **8.13** de CoqIde. No se garantiza el funcionamiento del proyecto en versiones anteriores o posteriores de CoqIde.

3. Herramientas

Para el desarrollo del proyecto, utilizamos algunas bibliotecas auxiliares de Coq. La más importante de estas fue la biblioteca `BinNat`[2] 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 de árboles Patricia

4.1. Tries binarios

Decidimos basarnos completamente en[1] 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.

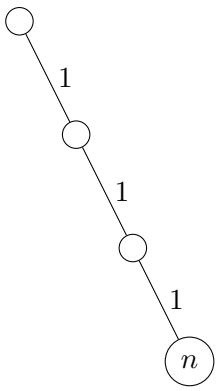
```
(* Constructor inteligente *)
Definition smartTrie1 (t1 t2: binTrie1) : binTrie1 :=
match t1, t2 with
| empty1, empty1 => empty1
| _, _ => trie1 t1 t2
end.
```

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`.

```
(* 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

4.5. Árboles Patricia

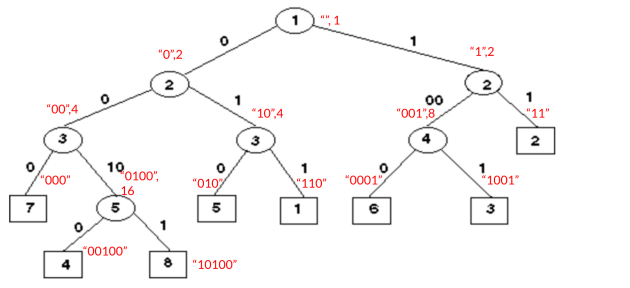
Nos encontramos muy cerca de los árboles Patricia. Solo nos falta hacer una optimización más a nuestros tries binarios.

El caso que nos falta optimizar es en el que todas las hojas tienen un prefijo en común. En este caso, la búsqueda recorrerá los bits del prefijo uno a la vez hasta llegar a un nodo del que salen todas las hojas. Esto se puede optimizar si nos evitamos recorrer los bits del prefijo. Como sabemos que hay un solo prefijo útil en el subárbol, podemos colapsar ese camino. La estructura bajo esta optimización es a lo que llamamos árbol Patricia. De nuevo, hay que cambiar nuestra estructura de datos, ahora para que los nodos internos guarden el prefijo que se ha leído hasta el momento. La nueva estructura está en el listado 8. Un ejemplo de un árbol Patricia está en la figura 3.

```
(* Árboles Patricia *)
Inductive patriciaTree: Type :=
| empty : patriciaTree
| leaf: N -> nat -> patriciaTree
| trie: N -> N -> patriciaTree -> patriciaTree -> patriciaTree.
```

Listado 8: Definición de árboles Patricia

Figura 3: Figura obtenida en [3] y modificada posteriormente. Los números en rojo representan los prefijos y bits que se guardan en cada nodo.



5. Metas del proyecto

Se plantearon las siguientes metas de manera concreta:

1. Definir tries binarios.
2. Definir árboles Patricia.
3. Definir la operación de búsqueda en árboles Patricia.
4. Definir la operación de inserción en árboles Patricia.
5. Demostrar corrección de inserción con respecto a búsquedas.
6. Definir operación de unión de dos árboles Patricia.
7. Demostrar corrección de la unión.
8. Definir una operación alternativa de búsqueda más óptima.
9. Demostrar equivalencia entre ambas operaciones de búsqueda.

6. Resultados

Todas las metas se lograron excepto por la meta 7. Las metas 1 y 2 se mencionaron en la sección 4, por lo que únicamente mencionaremos los resultados de las demás metas.

6.1. Meta 3

La implementación de la función de búsqueda se puede ver en el listado 9 y es practicamente una copia de la que se encuentra en[1]. La función `matchPrefix` se encarga de ver si, dados tres enteros k, p y m, k y p coinciden en los primeros $\log_2 m$ bits.

```
(* Nueva función de búsqueda
 * Igual a la del caso pasado, solo que ahora los nodos guardan el prefijo
 * común de sus hijos. *)
Fixpoint lookup (k: N) (t: patriciaTree) : option nat :=
match t with
| empty => None
| leaf j x => if (N.eqb j k) then Some x
              else None
| trie p m t0 t1 => if (negb (matchPrefix k p m)) then None
                   else if (zeroBit k m) then lookup k t0
                        else lookup k t1
end.
```

Listado 9: Función de búsqueda para árboles Patricia

La función `mask` es una máscara como en la definición usual, salvo porque en lugar de recibir un número de la forma $2^n - 1$, recibe uno de la forma 2^n y saca el predecesor de este.

Las funciones `matchPrefix`, `zeroBit` y `mask` están en el listado 10.

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

(* Función de máscara.
 * m es un entero de la forma 2^i *)
Definition mask (k m: N): N :=
N.land k (N.pred m).

(* Función que revisa si dos enteros tienen los primeros x bits iguales.
 * m es 2^x *)
Definition matchPrefix (k p m: N) : bool :=
N.eqb (mask k m) p.
```

Listado 10: Definiciones de operaciones binarias en Coq

6.2. Meta 4

La función de inserción también se obtuvo de[1] y está en el listado 11.

```
(* Función de inserción
 * Parámetros:
 * c - función a aplicar en caso de que la llave ya tenga elemento.
 * k - llave
 * x - número a insertar
 * t - árbol *)
Fixpoint insert (c : nat -> nat -> nat) (k: N) (x: nat) (t: patriciaTree) :=
match t with
| empty => leaf k x
| leaf j y => if (N.eqb j k) then leaf k (c x y)
              else join k j (leaf k x) t
| trie p m t0 t1 => if (matchPrefix k p m) then
                      if (zeroBit k m) then trie p m (insert c k x t0) t1
                      else trie p m t0 (insert c k x t1)
                    else join k p (leaf k x) t
end.
```

Listado 11: Función de inserción

6.3. Meta 5

Se logró satisfactoriamente demostrar la correción de la inserción con respecto a la búsqueda. Se demostró la corrección en dos casos: cuando el elemento insertado no estaba previamente en el árbol y cuando sí estaba ahí.

6.4. Meta 6

Esta meta se logró satisfactoriamente, aunque no de la manera que se esperaba. Debido a que la función de unión que se obtuvo del artículo no era recursiva simple, esta no se pudo definir usando `Fixpoint`. Investigando en internet se encontró un conjunto de tácticas denominadas `Program`[4], en donde se encuentra el comando `Program Fixpoint`, a partir del cual se pueden definir funciones con recursiones más complicadas que las aceptadas por `Fixpoint`.

Para definir una función con `Program Fixpoint`, basta con hacer lo siguiente:

1. Definir una medida que decrezca con cada paso de la recursión. En nuestro caso, utilizamos el número de nodos de los árboles como medida.

2. Probar que la recursión termina.

La definición de nuestra función con `Program Fixpoint` está en el listado 12. Tras definirla, tuvimos que probar una serie de obligaciones para convencer al asistente de que la recursión terminaba.

La función `br` es el constructor inteligente de árboles Patricia y está en el listado 13.

```
(* Función de combinación.
   Usamos la medida de número de nodos para decirle a Coq que los
   argumentos sí se decrementan
 * Parámetros:
 * c - función a aplicar en caso de que la llave ya tenga elemento.
 * s t - árboles *)
Program Fixpoint merge (c : nat -> nat -> nat) (s t: patriciaTree)
  {measure ((nodes s) + (nodes t))} : patriciaTree :=
match s, t with
| empty, _ => t
| _, empty => s
| leaf k x, _ => insert c k x t
| _, leaf k x => insert (swap c) k x t
| trie p m s0 s1, trie q n t0 t1 => if (N.eqb m n) && (N.eqb p q)
                                   (* Los árboles comparten prefijo *)
                                   then trie p m (merge c s0 t0) (merge c s1 t1)
                                   else if (N.ltb m n) && (matchPrefix q p m)
                                   (* Caso en que q contiene a p *)
                                        then if (zeroBit q m)
                                             then br p m (merge c s0 t) s1
                                             else br p m s0 (merge c s1 t)
                                        else if (N.ltb n m) && (matchPrefix p q n)
                                             then if (zeroBit p n)
                                                  then br q n (merge c s t0) t1
                                                  else br q n t0 (merge c s t1)
                                             else join p q s t
end.
```

Listado 12: Función de inserción

```
(* Nuevo constructor inteligente. *)
Definition br (p m: N) (p1 p2: patriciaTree) : patriciaTree :=
match p1, p2 with
| empty, _ => p2
| _, empty => p1
| _, _ => trie p m p1 p2
end.
```

Listado 13: Constructor inteligente de árboles Patricia

Este programa que a la vista de cualquier programador experimentado parece sencillo, en realidad causa dificultades para sistemas tan rigurosos como Coq. Es una de las formas en que podemos darnos cuenta de que la verificación formal aún tiene mucho hacia dónde crecer.

6.5. Meta 7

Esta meta no se logró debido a la poca experiencia con el uso de `Program`.

Al intentar demostrar propiedades con respecto a la función `merge`, nos topamos con que las definiciones escritas con `Program Fixpoint` no se pueden simplificar. Esto nos dejó como opción el desdoblar la definición. Desgraciadamente, parece que no era la forma correcta de resolver el problema, pues el desdoblar la definición nos da lo que está en el listado 14: un programa ilegible.

```
Fix_sub {_ : nat -> nat -> nat & {_ : patriciaTree & patriciaTree}}
  (MR lt
     (fun recarg : {_ : nat -> nat -> nat & {_ : patriciaTree & patriciaTree}} =>
      nodes (projT1 (projT2 recarg)) + nodes (projT2 (projT2 recarg))))
  merge_func_obligation_12
  (fun _ : {_ : nat -> nat -> nat & {_ : patriciaTree & patriciaTree}} => patriciaTree)
  (fun (recarg : {_ : nat -> nat -> nat & {_ : patriciaTree & patriciaTree}})
     (merge' : {recarg' : {_ : nat -> nat -> nat & {_ : patriciaTree & patriciaTree}}
                | nodes (projT1 (projT2 recarg')) + nodes (projT2 (projT2 recarg')) <
                  nodes (projT1 (projT2 recarg)) + nodes (projT2 (projT2 recarg))} ->
               patriciaTree) =>
   match
     projT1 (projT2 recarg) as t'
     return
   (t' = projT1 (projT2 recarg) ->
        projT2 (projT2 recarg) = projT2 (projT2 recarg) -> patriciaTree)
   with
   | empty =>
       fun (_ : empty = projT1 (projT2 recarg))
         (_ : projT2 (projT2 recarg) = projT2 (projT2 recarg)) =>
       projT2 (projT2 recarg)
   | leaf k x =>
       match
         projT2 (projT2 recarg) as s'
         return
           (leaf k x = projT1 (projT2 recarg) ->
            s' = projT2 (projT2 recarg) -> patriciaTree)
       with
       | empty =>
           fun (_ : leaf k x = projT1 (projT2 recarg))
             (_ : empty = projT2 (projT2 recarg)) => projT1 (projT2 recarg)
       | leaf k0 x0 =>
   fun (_ : leaf k x = projT1 (projT2 recarg))
             (_ : leaf k0 x0 = projT2 (projT2 recarg)) =>
           insert (projT1 recarg) k x (projT2 (projT2 recarg))
       | trie n n0 p p0 =>
           fun (_ : leaf k x = projT1 (projT2 recarg))
             (_ : trie n n0 p p0 = projT2 (projT2 recarg)) =>
```

```
        insert (projT1 recarg) k x (projT2 (projT2 recarg))
    end
| trie p m s0 s1 =>
    match
      projT2 (projT2 recarg) as s'
      return
        (trie p m s0 s1 = projT1 (projT2 recarg) ->
         s' = projT2 (projT2 recarg) -> patriciaTree)
    with
    | empty =>
        fun (_ : trie p m s0 s1 = projT1 (projT2 recarg))
          (_ : empty = projT2 (projT2 recarg)) => projT1 (projT2 recarg)
    | leaf k x =>
        fun (_ : trie p m s0 s1 = projT1 (projT2 recarg))
          (_ : leaf k x = projT2 (projT2 recarg)) =>
        insert (swap (projT1 recarg)) k x (projT2 (projT2 recarg))
    | trie q n t0 t3 =>
        fun (Heq_s : trie p m s0 s1 = projT1 (projT2 recarg))
          (Heq_t : trie q n t0 t3 = projT2 (projT2 recarg)) =>
        if (m =? n)%N && (p =? q)%N
        then
         trie p m
           (merge'
              (exist
                 (fun
                    recarg' : {_ : nat -> nat -> nat &
                                {_ : patriciaTree & patriciaTree}} =>
                  nodes (projT1 (projT2 recarg')) + nodes (projT2 (projT2 recarg')) <
                  nodes (projT1 (projT2 recarg)) + nodes (projT2 (projT2 recarg)))
                 (existT
                    (fun _ : nat -> nat -> nat => {_ : patriciaTree & patriciaTree})
                    (projT1 recarg)
                    (existT (fun _ : patriciaTree => patriciaTree) s0 t0))
                 (merge_func_obligation_1 (projT1 (projT2 recarg))
                    (projT2 (projT2 recarg))
                    (fun (c0 : nat -> nat -> nat) (s t : patriciaTree)
                       (recproof : nodes s + nodes t <
                                   nodes (projT1 (projT2 recarg)) +
                                   nodes (projT2 (projT2 recarg))) =>
                     merge'
                       (exist
                          (fun
                             recarg' : {_ : nat -> nat -> nat &
                                         {_ : patriciaTree & patriciaTree}} =>
                           nodes (projT1 (projT2 recarg')) +
```

```
                    nodes (projT2 (projT2 recarg')) <
                    nodes (projT1 (projT2 recarg)) +
                    nodes (projT2 (projT2 recarg)))
                   (existT
                      (fun _ : nat -> nat -> nat =>
                       {_ : patriciaTree & patriciaTree}) c0
                      (existT (fun _ : patriciaTree => patriciaTree) s t))
                   recproof)) p m s0 s1 q n t0 t3 Heq_s Heq_t)))
   (merge'
      (exist
         (fun
            recarg' : {_ : nat -> nat -> nat &
                       {_ : patriciaTree & patriciaTree}} =>
          nodes (projT1 (projT2 recarg')) + nodes (projT2 (projT2 recarg')) <
          nodes (projT1 (projT2 recarg)) + nodes (projT2 (projT2 recarg)))
         (existT
            (fun _ : nat -> nat -> nat => {_ : patriciaTree & patriciaTree})
            (projT1 recarg)
            (existT (fun _ : patriciaTree => patriciaTree) s1 t3))
         (merge_func_obligation_2 (projT1 (projT2 recarg))
            (projT2 (projT2 recarg))
            (fun (c0 : nat -> nat -> nat) (s t : patriciaTree)
               (recproof : nodes s + nodes t <
                           nodes (projT1 (projT2 recarg)) +
                           nodes (projT2 (projT2 recarg))) =>
             merge'
               (exist
                  (fun
                     recarg' : {_ : nat -> nat -> nat &
                                {_ : patriciaTree & patriciaTree}} =>
                   nodes (projT1 (projT2 recarg')) +
                   nodes (projT2 (projT2 recarg')) <
                   nodes (projT1 (projT2 recarg)) +
                   nodes (projT2 (projT2 recarg)))
                  (existT
                     (fun _ : nat -> nat -> nat =>
                      {_ : patriciaTree & patriciaTree}) c0
                     (existT (fun _ : patriciaTree => patriciaTree) s t))
                  recproof)) p m s0 s1 q n t0 t3 Heq_s Heq_t)))
else
 if (m <? n)%N && matchPrefix q p m
 then
  if zeroBit q m
  then
   br p m
```

```
    (merge'
       (exist
          (fun
             recarg' : {_ : nat -> nat -> nat &
                        {_ : patriciaTree & patriciaTree}} =>
           nodes (projT1 (projT2 recarg')) +
           nodes (projT2 (projT2 recarg')) <
           nodes (projT1 (projT2 recarg)) + nodes (projT2 (projT2 recarg)))
          (existT
             (fun _ : nat -> nat -> nat =>
              {_ : patriciaTree & patriciaTree})
             (projT1 recarg)
             (existT (fun _ : patriciaTree => patriciaTree) s0
                (projT2 (projT2 recarg))))
          (merge_func_obligation_3 (projT1 (projT2 recarg))
             (projT2 (projT2 recarg))
             (fun (c0 : nat -> nat -> nat) (s t : patriciaTree)
                (recproof : nodes s + nodes t <
                            nodes (projT1 (projT2 recarg)) +
                            nodes (projT2 (projT2 recarg))) =>
              merge'
                (exist
                   (fun
                      recarg' : {_ : nat -> nat -> nat &
                                 {_ : patriciaTree & patriciaTree}} =>
                    nodes (projT1 (projT2 recarg')) +
                    nodes (projT2 (projT2 recarg')) <
                    nodes (projT1 (projT2 recarg)) +
                    nodes (projT2 (projT2 recarg)))
                   (existT
                      (fun _ : nat -> nat -> nat =>
                       {_ : patriciaTree & patriciaTree}) c0
                      (existT (fun _ : patriciaTree => patriciaTree) s t))
                   recproof)) p m s0 s1 q n t0 t3 Heq_s Heq_t))) s1
else
 br p m s0
   (merge'
      (exist
         (fun
            recarg' : {_ : nat -> nat -> nat &
                       {_ : patriciaTree & patriciaTree}} =>
          nodes (projT1 (projT2 recarg')) +
          nodes (projT2 (projT2 recarg')) <
          nodes (projT1 (projT2 recarg)) + nodes (projT2 (projT2 recarg)))
         (existT
```

```
                (fun _ : nat -> nat -> nat =>
                 {_ : patriciaTree & patriciaTree})
                (projT1 recarg)
                (existT (fun _ : patriciaTree => patriciaTree) s1
                   (projT2 (projT2 recarg))))
             (merge_func_obligation_4 (projT1 (projT2 recarg))
                (projT2 (projT2 recarg))
                (fun (c0 : nat -> nat -> nat) (s t : patriciaTree)
                   (recproof : nodes s + nodes t <
                               nodes (projT1 (projT2 recarg)) +
                               nodes (projT2 (projT2 recarg))) =>
                 merge'
                   (exist
                      (fun
                         recarg' : {_ : nat -> nat -> nat &
                                    {_ : patriciaTree & patriciaTree}} =>
                       nodes (projT1 (projT2 recarg')) +
                       nodes (projT2 (projT2 recarg')) <
                       nodes (projT1 (projT2 recarg)) +
                       nodes (projT2 (projT2 recarg)))
                      (existT
                         (fun _ : nat -> nat -> nat =>
                          {_ : patriciaTree & patriciaTree}) c0
                         (existT (fun _ : patriciaTree => patriciaTree) s t))
                      recproof)) p m s0 s1 q n t0 t3 Heq_s Heq_t)))
else
 if (n <? m)%N && matchPrefix p q n
 then
  if zeroBit p n
  then
   br q n
     (merge'
        (exist
           (fun
              recarg' : {_ : nat -> nat -> nat &
                         {_ : patriciaTree & patriciaTree}} =>
            nodes (projT1 (projT2 recarg')) +
            nodes (projT2 (projT2 recarg')) <
            nodes (projT1 (projT2 recarg)) + nodes (projT2 (projT2 recarg)))
           (existT
              (fun _ : nat -> nat -> nat =>
               {_ : patriciaTree & patriciaTree})
              (projT1 recarg)
              (existT (fun _ : patriciaTree => patriciaTree)
                 (projT1 (projT2 recarg)) t0))
```

```
            (merge_func_obligation_5 (projT1 (projT2 recarg))
               (projT2 (projT2 recarg))
               (fun (c0 : nat -> nat -> nat) (s t : patriciaTree)
                  (recproof : nodes s + nodes t <
                              nodes (projT1 (projT2 recarg)) +
                              nodes (projT2 (projT2 recarg))) =>
                   merge'
                     (exist
                        (fun
                           recarg' : {_ : nat -> nat -> nat &
                                      {_ : patriciaTree & patriciaTree}} =>
                         nodes (projT1 (projT2 recarg')) +
                         nodes (projT2 (projT2 recarg')) <
                         nodes (projT1 (projT2 recarg)) +
                         nodes (projT2 (projT2 recarg)))
                        (existT
                           (fun _ : nat -> nat -> nat =>
                            {_ : patriciaTree & patriciaTree}) c0
                           (existT (fun _ : patriciaTree => patriciaTree) s t))
                        recproof)) p m s0 s1 q n t0 t3 Heq_s Heq_t))) t3
else
 br q n t0
   (merge'
      (exist
         (fun
            recarg' : {_ : nat -> nat -> nat &
                       {_ : patriciaTree & patriciaTree}} =>
          nodes (projT1 (projT2 recarg')) +
          nodes (projT2 (projT2 recarg')) <
          nodes (projT1 (projT2 recarg)) + nodes (projT2 (projT2 recarg)))
         (existT
            (fun _ : nat -> nat -> nat =>
             {_ : patriciaTree & patriciaTree})
            (projT1 recarg)
            (existT (fun _ : patriciaTree => patriciaTree)
               (projT1 (projT2 recarg)) t3))
         (merge_func_obligation_6 (projT1 (projT2 recarg))
            (projT2 (projT2 recarg))
            (fun (c0 : nat -> nat -> nat) (s t : patriciaTree)
               (recproof : nodes s + nodes t <
                           nodes (projT1 (projT2 recarg)) +
                           nodes (projT2 (projT2 recarg))) =>
             merge'
               (exist
                  (fun
```

```
                                    recarg' : {_ : nat -> nat -> nat &
                                               {_ : patriciaTree & patriciaTree}} =>
                                   nodes (projT1 (projT2 recarg')) +
                                   nodes (projT2 (projT2 recarg')) <
                                   nodes (projT1 (projT2 recarg)) +
                                   nodes (projT2 (projT2 recarg)))
                                  (existT
                                     (fun _ : nat -> nat -> nat =>
                                      {_ : patriciaTree & patriciaTree}) c0
                                     (existT (fun _ : patriciaTree => patriciaTree) s t))
                                  recproof)) p m s0 s1 q n t0 t3 Heq_s Heq_t)))
            else join p q (projT1 (projT2 recarg)) (projT2 (projT2 recarg))
      end
   end eq_refl eq_refl)
  (existT (fun _ : nat -> nat -> nat => {_ : patriciaTree & patriciaTree}) c
     (existT (fun _ : patriciaTree => patriciaTree) t1 t2))
```

Listado 14: Código ilegible

6.6. Metas 8 y 9

Como no logramos la meta 7, decidimos cambiarla por otra tarea. En el artículo se menciona una definición alternativa de `lookup` que es un poco más eficiente. La tarea que nos dimos fue definir esta función y probar su equivalencia con la otra función de búsqueda. La definición de la función está en el listado 15.

```
(* Función de búsqueda equivalente a la primera pero un poco más eficiente. *)
Fixpoint lookup2 (k: N) (t:patriciaTree) :option nat :=
match t with
| empty => None
| leaf j x => if (N.eqb j k) then Some x
              else None
| trie p m t0 t1 => if (zeroBit k m) then lookup2 k t0
                         else lookup2 k t1
end.
```

Listado 15: Función de búsqueda más eficiente

La meta se logró de manera satisfactoria, aunque por desgracia, tuvimos que admitir algunos lemas sobre números binarios por falta de tiempo. Considerando que nuestro archivo de propiedades sobre números binarios ya sobrepasa las 500 líneas, es de nuestra opinión que haber demostrado toda las propiedades habría sido un proyecto por sí mismo; por esto, nos sentimos satisfechos a pesar de no haber demostrado todo.

7. Conclusiones

Se lograron satisfactoriamente la mayoría de las metas. Para un futuro, sería bueno aprender a utilizar `Program` y terminar las demostraciones de números binarios; aún así, el trabajo que se hizo está bastante completo.
Con este proyecto nos quedamos con el aprendizaje de que realizar un proyecto de un ejemplo real, por más pequeño que parezca, es una gran tarea. La verificación formal está en pañales y con buena razón; es difícil avanzar en el trabajo. Sin embargo, creemos que poco a poco, mientras haya dedicación e ideas, habrá un futuro en el que el área de verificación formal prolifere.

Referencias

[1] C. Okasaki and A. Gill, "Fast Mergeable Integer Maps," in *In Workshop on ML*, 1998, pp. 77–86.

[2] "Standard Library | The Coq Proof Assistant." [Online]. Available: https://coq.inria.fr/library/Coq.NArith.BinNat.html

[3] "Unit SS, Part A: PAT, PAT Trees." [Online]. Available: https://fox.cs.vt.edu/cs5604/cs5604cnSS/SS-a2.html

[4] "Program — Coq 8.13.0 documentation." [Online]. Available: https://coq.inria.fr/refman/addendum/program.html