1 Objetivo

Implementar la estructura de datos zipper[**?**] usando el lenguaje gallina en el asistente de pruebas coq.

2 Zipper sobre arboles binarios

2.1 Definiciones estructurales y propiedades de igualdad

Para el proyecto se utilizaron los módulos de listas y peano que el asistente provee. El primer elemento que se definió es el árbol, ya que es la estructura sobre la cual operará el zipper. El árbol se define de manera inductiva:

```
Inductive tree (X:Type) : Type :=
| nilTree : tree X
| branch : tree X → X → tree X → tree X.
```

De aquí en adelante para un árbol no nulo o *branch*, se le llamara raíz al segundo elemento que requiere el constructor *branch*, hijo izquierdo al primer elemento e hijo derecho al tercero. Por conveniencia una hoja(*leaf*) se define como un árbol no nulo cuyos hijos son *nilTree*.

```
Definition leaf {X:Type} (a:X): tree X :=
branch X (nilTree X) a (nilTree X).
```

Un elemento necesario para la estructura zipper es el contexto o path, el cual contiene los elementos ya visitados en el árbol. Para este proyecto se definió de manera inductiva una estructura que consta de 3 elementos, un contexto *top*, un contexto derecho *rightContext* que contiene la raíz de un árbol, el hijo derecho de dicha raíz, además de un contexto padre, finalmente se define un contexto izquierdo *leftContext* que de manera simétrica al derecho contiene un elemento raíz, su hijo izquierdo y un contexto padre.

```
Inductive context (X:Type) : Type :=
| top: context X
| rightContext: X → tree X → context X → context X
| leftContext: tree X → X → context X → context X.
```

Una vez definidos los árboles y contextos, es posible definir el zipper como la unión de estos dos. La función del contexto en el árbol es la de contener a los predecesores del árbol.

```
Inductive zipper (X:Type) :Type :=
| hole: tree X → context X → zipper X.
```

De ahora en adelante se le llamara *foco* a la raíz del árbol que compone un zipper. El foco de un zipper es el nodo del árbol al cual tenemos acceso directo.

2.2 Funciones sobre zippers

Para este proyecto, se implementaron 4 funciones de navegación, que son suficientes para moverse en el árbol.

- **go_up:** Mueve el foco hacia el padre del foco actual.
- **go_left:** Se mueve hacia el hijo izquierdo del foco.
- **go_right:** Se mueve hacia el hijo derecho del foco.
- **go_sibling:** Se mueve hacia el otro hijo, es decir si el foco esta sobre el hijo derecho se mueve al izquierdo y viceversa.

La función *go_up* funciona bajo las siguientes reglas: Si el zipper de entrada está compuesto por un contexto de tipo *top* entonces regresa el mismo zipper de entrada, si el contexto presente el zipper es un contexto derecho entonces significa que el árbol enfocado es hijo izquierdo de los elementos del contexto derecho, por

lo que se reconstruye el árbol y el nuevo contexto será el padre del contexto derecho antes mencionado, de manera similar pero simétrica para el contexto izquierdo.

```
Definition go_up {X:Type}(z :zipper X): zipper X:=
 match z with
   |hole _ top ⇒ z
   |hole l (rightContext t r c)⇒ hole X (branch X l t r) c
   |hole r (leftContext l t c)⇒ hole X (branch X l t r) c
 end.
```

Para las funciones de navegación izquierda y derecha, si el árbol enfocado es un *nilTree* entonces ambas funciones regresan como salida el mismo zipper de entrada, en cualquier otro caso navegan al hijo izquierdo o derecho del árbol según corresponda.

```
Definition go_left {X:Type}(z :zipper X): zipper X:=
 match z with
   |hole nilTree _ ⇒ z
   |hole (branch l t r) c ⇒ hole X l (rightContext X t r c)
 end.

Definition go_right {X:Type}(z :zipper X): zipper X:=
 match z with
   |hole nilTree _ ⇒ z
   |hole (branch l t r) c ⇒ hole X r (leftContext X l t c)
 end.
```

La función *go_sibling* simplifica moverse de manera horizontal en el arbol. De lo contrario se requeriría primero aplicar una operación *go_up* y después una operación *go_left* o *go_right* 'para moverse hacia el otro hijo de un foco.

```
Definition go_sibling {X:Type}(z :zipper X): zipper X:=
 match z with
   |hole _ top ⇒ z
   |hole l (rightContext t r c)=¿ hole X r (leftContext X l t c)
   |hole r (leftContext l t c)=¿ hole X l (rightContext X t r c)
 end.
```

Para concatenar varias funciones de navegación, así como otras funciones que operan sobre zippers y devuelven un zipper como resultado se definió la propiedad *apply_functions* que, dada una lista de funciones sobre zippers, aplica cada operación de manera secuencial.

```
Fixpoint apply_functions {X:Type} (l :list (zipper X → zipper X))(z :zipper X) : zipper X :=
match l with
| nil ⇒ z
| a::l0 ⇒ apply_functions l0 (a z)
end.
```

Una vez definidas las funciones de navegación es natural definir una función que nos permita operar sobre el árbol enfocado, en este caso la función *update* actualiza el valor de la raíz del árbol enfocado, si este último es un *nilTree* entonces no se realiza ningún cambio. También se define una función que extraiga el valor de la raíz del árbol enfocado.

```
Definition update {X:Type}(n :X)(z :zipper X) : zipper X :=
 match z with
   |hole nilTree _ ⇒ z
   |hole (branch l t r) c ⇒ hole X (branch X l n r) c
end.

Definition get_focus {X:Type}(t :zipper X) : option X :=
match t with
   |hole nilTree _ ⇒ None
   |hole (branch _ t _) _ ⇒ Some t
```

```
end.
```

Una vez adquirida la habilidad de navegar y editar de manera simple en el árbol a través del zipper, se define una función que transforme un árbol a un zipper.

```
Definition tree_to_zipper {X:Type}(t : tree X) : zipper X := hole X t (top X).
```

Evidentemente es también necesaria una función que transforme un zipper en un árbol, el caso trivial en el que el contexto del zipper es de tipo *top* se resuelve devolviendo el árbol enfocado. En cualquier otro caso se debe tomar en cuenta que el zipper puede estar en un estado en el cual cualquier subárbol del árbol original este enfocado, entonces la función que se trata de definir debe realizar varias operaciones del tipo *go_up* hasta que el contexto del zipper sea tipo *top* y se resuelva el caso trivial. como primer intento se puede traducir la idea antes mencionada a una función Fixpoint de gallina.

```
Fixpoint zipper_to_tree_erroneo {X:Type}(z : zipper X) : tree X :=
match z with
 |hole t top ⇒ t
 |hole t c ⇒ zipper_to_tree_erroneo(go_up(t))
end.
```

En la función anterior el asistente devuelve el error "Error: Cannot guess decreasing argument of fix." esto es debido a que no es evidente que cada aplicación de la función *go_up* nos acerque más al caso trivial. Una manera de resolver esto es utilizando Program Fixpoint el cual requiere una función que mida el hecho de que cada recursión de la función nos acerca al caso trivial, por ejemplo, en listas mediría la longitud de las listas y cada recursión disminuye esta longitud, también esta notación requiere mostrar que la función es una recursión bien fundada. Para evitar estos inconvenientes se procede definiendo una función que opera sobre un contexto y un árbol, la función contempla 3 casos, el primero si la entrada es de tipo *top* entonces devuelve el árbol de entrada, si el contexto es de tipo derecho entonces une el árbol de entrada al hijo derecho, la función se llama recursivamente ahora utilizando este nuevo árbol como entrada y el contexto padre como entrada, así hasta llegar al caso trivial, se procede de manera similar y simétrica para el contexto izquierdo.

```
Fixpoint tree_from_context {X:Type} (t : tree X)(c : context X) : tree X :=
match c with
   |top ⇒ t
   |rightContext f r c1 ⇒ tree_from_context (branch X t f r) c1
   |leftContext l f c1 ⇒ tree_from_context (branch X l f t) c1
end.
```

Esta función es lo que se necesita para definir la operación que transforme zippers a árboles. Y se define de manera directa:

```
Definition zipper_to_tree {X:Type}(z : zipper X) : tree X :=
match z with
 |hole t c ⇒ tree_from_context t c
end.
```

Como ejercicio, derivado de lo anterior se definen dos funciones para medir la profundidad(nivel) en el que se encuentra operando el zipper.

```
Fixpoint context_depth {X:Type}(c :context X) : nat :=
match c with
    |top ⇒ 0
    |rightContext _ _ c1 ⇒ S (context_depth(c1))
    |leftContext _ _ c2 ⇒ S (context_depth(c2))
end.
```

```
Fixpoint zipper_depth {X:Type} (z : zipper X) : nat :=
match z with
 |hole _ c ⇒ context_depth(c)
end.
```

Para demostrar que la aplicación de las funciones de navegación no modifica la estructura del zipper y del árbol sobre el que operan, se definió la propiedad *unaltering_function* de la siguiente manera:

`Definition` *unaltering_function* {*X*:`Type`} (*f*:*zipper X* → *zipper X*) :`Prop` :=
∀ *z*:*zipper X*, *move_zipper_top*(*f z*) = *move_zipper_top z*.

2.3 Proposiciones

Las primeras proposiciones demostradas son aquellas que indican los inversos de las funciones de navegación, esto es importante ya que así conocemos con exactitud como revertir la operación aplicada al zipper y regresar al punto inicial.

`Proposition` *inverse_go_left*: ∀ *X*:`Type`, ∀ *c*:`context` *X*,∀ *t*:*tree X*,
(˜ ((*nilTree X*) = *t*)) → (*go_up*(*go_left*((*hole X t c*)))) = (*hole X t c*).

`Proposition` *inverse_go_right*: ∀ *X*:`Type`, ∀ *c*:`context` *X*,∀ *t*:*tree X*,
(˜ ((*nilTree X*) = *t*)) → (*go_up*(*go_left*((*hole X t c*)))) = (*hole X t c*).

`Proposition` *inverse_go_sibling*: ∀ *X*:`Type`, ∀ *c*:`context` *X*,∀ *t*:*tree X*,
(˜ ((*nilTree X*) = *t*)) → (*go_sibling*(*go_sibling*((*hole X t c*)))) = (*hole X t c*).

Se demostró que las funciones de navegación no alteran la estructura del árbol sobre el que operan, de igual manera se comprobó que la aplicación de varias de estas funciones no altera al árbol.

`Proposition` *is_unaltering*: ∀ *X*:`Type`,∀ *f*:*zipper X* → *zipper X*,
(∀ *z*: *zipper X*, *move_zipper_top z* = *move_zipper_top* (*f z*)) ↔ *unaltering_function f*.

`Proposition` *go_left_is_unaltering*: ∀ *X*:`Type`, *unaltering_function* (@*go_left X*).

`Proposition` *go_right_is_unaltering*: ∀ *X*:`Type`, *unaltering_function* (@*go_right X*).

`Proposition` *go_up_is_unaltering*: ∀ *X*:`Type`, *unaltering_function* (@*go_up X*).

`Proposition` *go_sibling_is_unaltering*: ∀ *X*:`Type`, *unaltering_function* (@*go_sibling X*).

`Proposition` *chaining_unaltering_functions*: ∀ *X*:`Type`, ∀ *operations*:*list* (*zipper X* → *zipper X*), ∀ *a*:*zipper X* → *zipper X*,
(*unaltering_function a* ∧ *unaltering_function* (*apply_functions operations*)) → *unaltering_function* (*apply_functions* (*a* :: *operations*)).

`Proposition` *composition_of_unaltering_is_unalterng*: ∀ *X*:`Type`,∀ *operations*:*list* (*zipper X* → *zipper X*),
(*all_in unaltering_function operations*) → *unaltering_function* (*apply_functions operations*).

`Proposition` *apply_unaltering_functions*: ∀ *X*:`Type`,∀ *operations*:*list* (*zipper X* → *zipper X*),
(*all_in unaltering_function operations*) → (∀ *t*:*tree X*,*tree_to_zipper t* = *move_zipper_top*(*apply_functions operations* (*tree_to_zipper t*))).

References

[1] *HUET, G.* (1997). The Zipper. Journal of Functional Programming, 7(5), 549–554. https://doi.org/10.1017/s0956796897002864