Ejercicio 1

1. (a)

El módulo A exporta el tipo abstracto `data S`, lo que significa que no exporta sus constructores. Además exporta `data T` contenido en el módulo B y el constructor D del mismo.

1. (b)

```
1 module A (S (), module B) where
2 import Prelude ()
3 import B (T (..))
4 data S = C | D
```

La segunda línea indica que no se debe importar ninguna función del módulo `Prelude`, por lo tanto los nombres que están al alcance son `data T` y el constructor `D`, los cuales son definidos en el módulo B, y `data S` y sus constructores C y D.

Los nombres cualificados están en el alcance del cuerpo.

El constructor D es el único nombre ambiguo. Esto se debe a que el módulo B exporta el constructor D de `data T` y dentro del módulo A existe un `data S` que contiene el mismo constructor. Para evitar esto se podría hacer un import con la opción de qualified y salir de la ambigüedad.

Ejercicio 2

2. (a)

Al intentar compilar el módulo B se obtiene el siguiente error

```
B.hs:3:8: error: Not in scope: type constructor or class ‘T’
  |
3 | baz :: T -> Int
  |        ^
```

Esto ocurre porque el módulo A no exporta `data T`, por lo tanto al intentar utilizarlo en B el compilador devuelve este mensaje de error.

2. (b)

Para poder solucionar este problema se puede comentar la línea que define el tipo de la función `baz` de esta forma el compilador infiere el tipo correctamente. Este problema se puede solucionar también si el módulo A exportara `data T` como tipo abstracto, de forma que el módulo B sepa que existe el tipo T, aunque no tenga forma de instanciarlo.

Ejercicio 3

```
*Main> quickCheck prop_f
+++ OK, passed 100 tests.
```

El resultado indica que en las cien pruebas, el resultado de prop_f es *true*, lo cual parece extraño, dado que significaría que el inverso de todas las tiras que fueron testeadas, fue igual a la tira.

Utilizando el comando `verboseCheck` se puede observar que todas las tiras no vacías están formadas solamente por *Units*, por lo que el inverso siempre será igual a la tira original.

```
ghci> verboseCheck prop_f
Passed:
[]

Passed:
[()]

Passed:
[]

Passed:
[(),()]
...
```

Esto ocurre debido a que QuickCheck debe generar instancias del tipo más general posible, en este caso `Eq a => [a]` y esto lo hace utilizando *Unit*.

Se pueden utilizar *type signatures* para elegir el tipo a utilizar en las pruebas, por ejemplo ejecutando `quickCheck (prop_f :: [Int] -> Bool)` se obtiene:

```
*Main> quickCheck (prop_f :: [Int] -> Bool)
*** Failed! Falsified (after 4 tests and 4 shrinks):
[0,1]
```

Lo cual es coherente.

Ejercicio 4

Ejecutando `quickCheck (\xs -> collect (size xs) (prop_size))` se puede observar que alrededor del 20% de los árboles generados son de largo 0, esto es acorde al código dado que uno de cada cinco llamadas a `arbitraryTree` resultan en un árbol `Empty`, el resto son de largo aleatorio (menor a 100 en este caso).

Si se aumenta la probabilidad de generar un nodo en el árbol arbitrario esto resulta en un menor número de arboles vacíos y se generan árboles de mayor tamaño, aunque el tamaño de los mismos siempre es acotado por la función `sized`, esto se ve claramente al elegir una probabilidad extremadamente pequeña para generar un nodo vacío, ya que los árboles resultan de un tamaño máximo de alrededor de 130.

Al revisar los árboles generados se puede ver además que los valores de los nodos son *units*, en caso de fijar el tipo a `Int`, los valores de los nodos son aleatorios.

Ejercicio 5

Dado que una propiedad QuickCheck es nada mas que una función que retorna un Boolean, entonces la función definida en la parte a es una propiedad QuickCheck que verifica si una lista es un palíndromo.[1]

Para asegurar la ejecución de 100 testeos exitosos basta con utilizar `quickCheckWith` en vez de `quickCheck` pasando un valor muy grande en el parámetro `maxDiscardRatio` (por ejemplo 10000), el mismo indica cual es el número de tests que deben fallar por cada test exitoso para terminar la prueba.

Para realizar 200 testeos exitosos se debe además utilizar el parámetro `maxSuccess` que fija la cantidad de testeos exitosos necesarios para terminar la prueba.

[1] Test.QuickCheck docs

Ejercicio 6

6. (a) $\forall(xs :: [a]).xs +\!\!+ [\,] \equiv xs$

Primero se probará el paso base donde xs es la lista vacía, es decir $xs = [\,]$.

$$xs +\!\!+ [\,] \equiv xs \tag{1}$$

Dado que $xs = [\,]$:

$$[\,] +\!\!+ [\,] \equiv [\,] \tag{2}$$

Y por definición de $+\!\!+$

$$[\,] \equiv [\,] \tag{3}$$

Para el paso inductivo se utilizara la hipótesis inductiva

$$xs +\!\!+ [] \equiv xs$$

donde xs :: [a].

Por lo tanto se probará que la propiedad se cumple para una lista $(x : xs)$, tal que $x :: a$ y $xs :: [a]$.

$$x : xs +\!\!+ [\,] \equiv x : xs \tag{4}$$

Por definición de $+\!\!+$ se sabe que $x : xs +\!\!+ [\,] = x : (xs +\!\!+ [\,])$

$$x : (xs +\!\!+ []) \equiv x : xs \tag{5}$$

Como $xs +\!\!+ [\,] \equiv xs$ se puede simplificar:

$$x : xs \equiv x : xs \tag{6}$$

Concluyendo entonces que $\forall(xs :: [a]).xs +\!\!+ [\,] \equiv xs$

6. (b) $\forall(xs :: [a]).[\,] +\!\!+ xs \equiv xs$

Esta demostración es bastante similar respecto a la anterior, por lo tanto se omitirán algunos pasos. Se tomará xs = [ ] como paso base.

$$[\,] +\!\!+ [\,] \equiv [\,] \xrightarrow{Def.+\!\!+} [\,] \equiv [\,] \tag{7}$$

Para el paso inductivo se utilizará el mismo razonamiento. Como hipótesis inductiva se tiene que la propiedad se cumple para xs, donde xs :: [a].

$$[\,] +\!\!+ x : xs \equiv x : xs \xrightarrow{Def.+\!\!+} x : xs \equiv x : xs \tag{8}$$

Por lo tanto se concluye que $\forall\,(xs :: [a])\,.\,[\,] +\!\!+ xs \equiv xs$

6. (c) $\forall(xs :: [a])(ys :: [a])(zs :: [a]).xs +\!\!+ (ys +\!\!+ zs) \equiv (xs +\!\!+ ys) +\!\!+ zs$

Proof. Tomando $xs = [\,]$, ys :: [a] y zs :: [a].

$$
\begin{aligned}
[\,] +\!\!+ (ys +\!\!+ zs) &\equiv ([\,] +\!\!+ ys) +\!\!+ zs \\
ys +\!\!+ zs &\equiv ([\,] +\!\!+ ys) +\!\!+ zs && \text{(Por b)} \\
ys +\!\!+ zs &\equiv ys +\!\!+ zs && \text{(Por b)}
\end{aligned}
$$

□

Por lo tanto se cumple esta propiedad para el paso base.
Ahora para el paso inductivo se debe probar que:

$$\forall(x :: a)(xs :: [a])(ys :: [a])(zs :: [a]).(x : xs) +\!\!+ (ys +\!\!+ zs) \equiv ((x : xs) +\!\!+ ys) +\!\!+ zs$$

Tomando entonces como hipótesis inductiva:

$$(xs :: [a])(ys :: [a])(zs :: [a]).xs +\!\!+ (ys +\!\!+ zs) \equiv (xs +\!\!+ ys) +\!\!+ zs$$

Proof.

$$
\begin{aligned}
(x : xs) +\!\!+ (ys +\!\!+ zs) &\equiv ((x : xs) +\!\!+ ys) +\!\!+ zs \\
x : (xs +\!\!+ (ys +\!\!+ zs)) &\equiv ((x : xs) +\!\!+ ys) +\!\!+ zs && \text{(Def. } +\!\!+\text{)} \\
x : ((xs +\!\!+ ys) +\!\!+ zs) &\equiv ((x : xs) +\!\!+ ys) +\!\!+ zs && \text{(Por hipótesis inductiva)} \\
x : ((xs +\!\!+ ys) +\!\!+ zs) &\equiv (x : (xs +\!\!+ ys)) +\!\!+ zs && \text{(Def. } +\!\!+\text{)} \\
x : ((xs +\!\!+ ys) +\!\!+ zs) &\equiv x : ((xs +\!\!+ ys) +\!\!+ zs) && \text{(Def. } +\!\!+\text{)}
\end{aligned}
$$

□

Por lo tanto, se puede concluir que la propiedad es verdadera.

6. (d) $\forall(xs :: [a])(ys :: [a]).reverse\ (xs +\!\!+ ys) \equiv reverse\ ys +\!\!+ reverse\ xs$

Paso Base:

Proof. Tomando $xs = [\,]$, ys :: [a]

$$
\begin{aligned}
reverse\ ([\,] +\!\!+ ys) &\equiv reverse\ ys +\!\!+ reverse\ [\,] \\
reverse\ ys &\equiv reverse\ ys +\!\!+ reverse\ [\,] && \text{(Por b)} \\
reverse\ ys &\equiv reverse\ ys +\!\!+ [\,] && \text{(Def. reverse)} \\
reverse\ ys &\equiv reverse\ ys && \text{(Por a)}
\end{aligned}
$$

□

Para el paso inductivo se debe probar que:

$$reverse\ ((x : xs) +\!\!+ ys) \equiv reverse\ ys +\!\!+ reverse\ (x : xs)$$

Hipótesis inductiva:

$$reverse\ (xs +\!\!+ ys) \equiv reverse\ ys +\!\!+ reverse\ xs$$

Proof.

$reverse\ ((x : xs) +\!\!+ ys) \equiv reverse\ ys +\!\!+ reverse\ (x : xs)$	
$reverse\ (x : (xs +\!\!+ ys) \equiv reverse\ ys +\!\!+ reverse\ (x : xs)$	(Def. $+\!\!+$)
$reverse\ (xs +\!\!+ ys) +\!\!+ [x] \equiv reverse\ ys +\!\!+ reverse\ (x : xs)$	(Def. reverse)
$reverse\ (xs +\!\!+ ys) +\!\!+ [x] \equiv reverse\ ys +\!\!+ reverse\ xs +\!\!+ [x]$	(Def. reverse)
$reverse\ (xs +\!\!+ ys) +\!\!+ [x] \equiv reverse\ (xs +\!\!+ ys) +\!\!+ [x]$	(Utilizando hipótesis inductiva)

□

Por lo tanto, se puede concluir que la propiedad es verdadera.

6. (e) $\forall(xs :: [a]).(palindromo\ xs \implies palindromo\ (xs +\!\!+ xs))$

Por definición de palíndromo sabemos que: $palindromo\ xs \implies xs = reverse\ xs$, por lo tanto se debe demostrar que $\forall(xs :: [a])\ .\ (palindromo\ xs \implies xs +\!\!+ xs \equiv reverse(xs +\!\!+ xs))$.

Proof.

$xs +\!\!+ xs \equiv reverse\ (xs +\!\!+ xs)$	
$xs +\!\!+ xs \equiv (reverse\ xs) +\!\!+ (reverse\ xs)$	(Por d)
$xs +\!\!+ xs \equiv xs +\!\!+ xs$	(xs es palíndromo)

□

Ejercicio 7

never executed always true always false

```
1  data Arbol a = Vacio | Nodo (Arbol a) a (Arbol a)
2
3  genera = fst $ generaAux 0
4    where
5      generaAux n =
6        let (l, n') = generaAux (n + 1)
7            (r, n'') = generaAux n'
8         in (Nodo l n r, n'')
9
10 recorreL (Nodo l x _) = x : recorreL l
11 recorreL Vacio = []
12
13 recorreR (Nodo _ x r) = x : recorreR r
14 recorreR Vacio = []
15
16 recorre (Nodo l x r) = recorre l ++ [x] ++ recorre r
17 recorre Vacio = []
18
19 take' _ [] = []
20 take' n (x : xs)
21   | n > 0 = x : take' (n - 1) xs
22   | otherwise = []
23
24 main = do
25   print (take' 5 . tail . recorreL $ genera)
26   print (head $ zip (recorreL genera) (recorreR genera))
```

El reporte indica que hay partes de código que nunca son ejecutadas, la función `genera` toma el primer elemento del par generado por la ejecución de `generaAux 0`, la cual genera un árbol infinito que crece hacia la izquierda sin llegar a generar ninguna rama derecha, esto se debe a que la misma no tiene paso base y cada llamada recursiva genera otra sin llegar a ejecutar `generaAux n'` en ningún momento, como consecuencia, `r` y `n''` nunca son generados. Esto no es un problema al momento de ejecutar el programa dado que en ningún momento se recorren las ramas derechas.

Se puede ver que la función `main` primero llama a `recorreL` la cual no necesita de ramas derechas, y luego ejecuta `head $ zip (recorreL genera) (recorreR genera)` por lo que solo lee el valor del primer nodo y no llega a generar ninguna rama, por lo cual `recorreR` no realiza ninguna llamada recursiva. Además `genera` es incapaz de generar un árbol vacío, por lo cual ninguna de las funciones que recorren el árbol resultan en una lista vacía y `take'` nunca ejecuta el caso en que la lista es vacía.

Por otro lado la función `recorre` nunca es llamada en `main` por lo que la linea 16 y 17 nunca son ejecutadas, además `otherwise` siempre se evalúa a true y HPC lo marca como tal. [2]

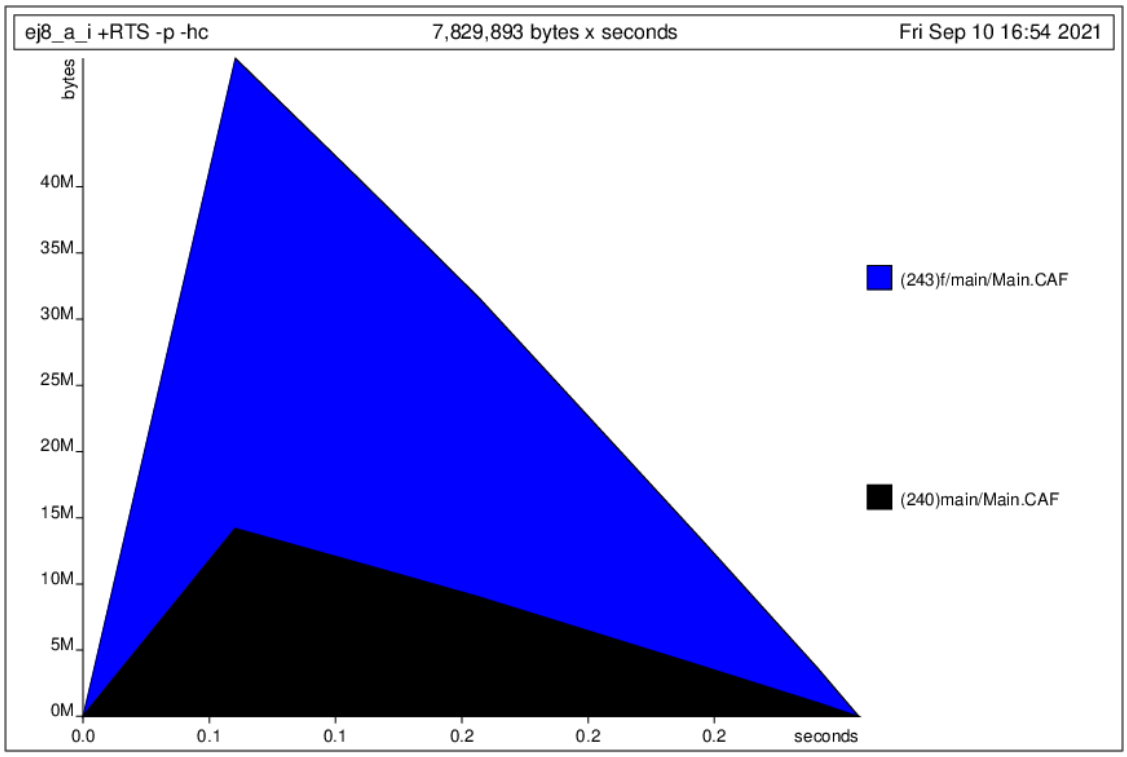
[2] GHC repository, issue #3175

Ejercicio 8

8. (a) `main = print $ f [1..N]`

8. (a) i. `f = foldl (flip (:)) []`

$N = 100000$



En color azul se muestra el heap que ocupa la función `f` llamada por main y en negro `main`.

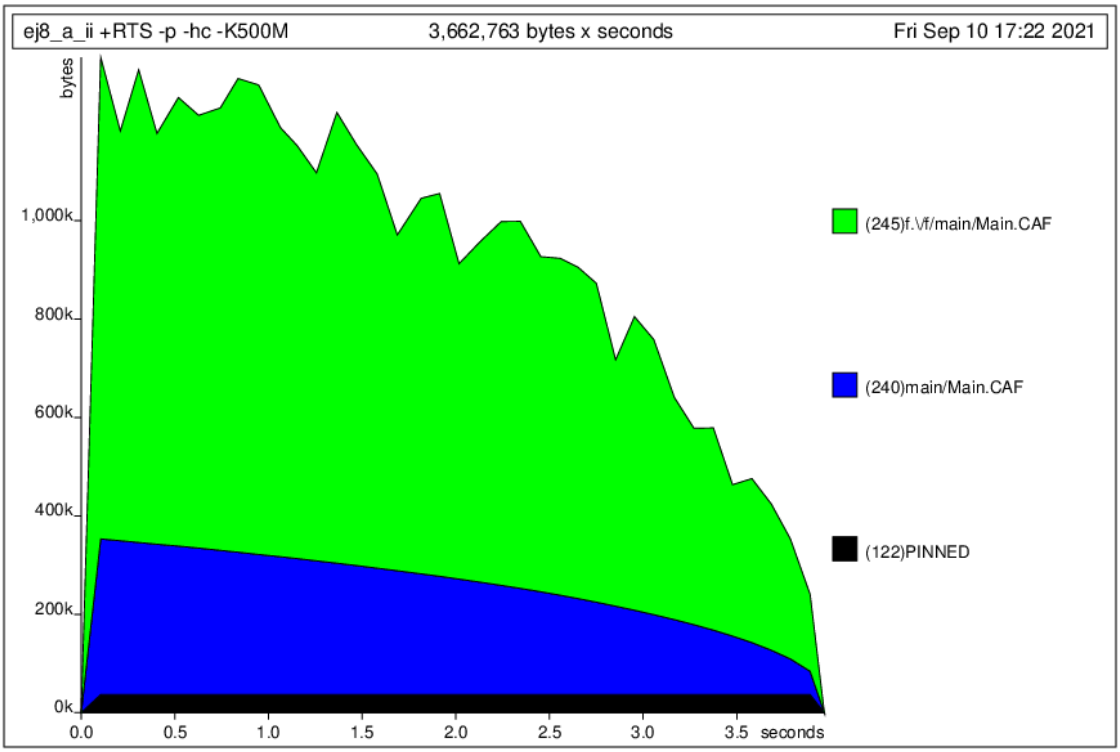
`f` invierte una lista y `main` la imprime, por lo tanto, como `print` necesita el primer elemento de la lista para comenzar a imprimir, lo que ocurre es que se debe computar por completo `f` antes de comenzar, incluyendo todos los `flip (:)` debido a la forma en la que quedan los paréntesis:

```
1 -- Resultado de foldl
2 ((...([] `(flip (:)` 1) `(flip (:))` ...) `(flip (:))` n)
```

Como resultado, la estructura se va generando hasta llegar al pico, donde ocupa alrededor de 50 MBytes y en ese momento se comienza a consumir la misma y a liberar la memoria, por ello la ascendencia y descendencia fuertemente marcadas en el gráfico.

8. (a) ii. `f = foldr (\x r -> r ++ [x]) []`

$N = 20000$



En este caso, debido a que la cantidad de memoria utilizada es menor, aparece otra componente llamada *PINNED*, en términos simples esta es memoria que puede ser coleccionada por el GC, pero que se indica que no debe ser eliminada.[3] En general es memoria del *runtime* del sistema.[4]

Al igual que en el caso anterior, dado que se elige el primer elemento y f la invierte, entonces se debe generar toda la estructura del `foldr`, pero a diferencia del anterior, ahora no se debe generar la totalidad de la lista para poder consumirla:

```
1 -- Resultado de foldr
2 ((...([] ++ [n]) ++ ...) ++ [1])
```

Por lo tanto, la lista se va generando a medida que se va consumiendo y el *Garbage Collector* la va liberando periódicamente, a medida que es consumida, a esto se deben los picos que aparecen en verde.

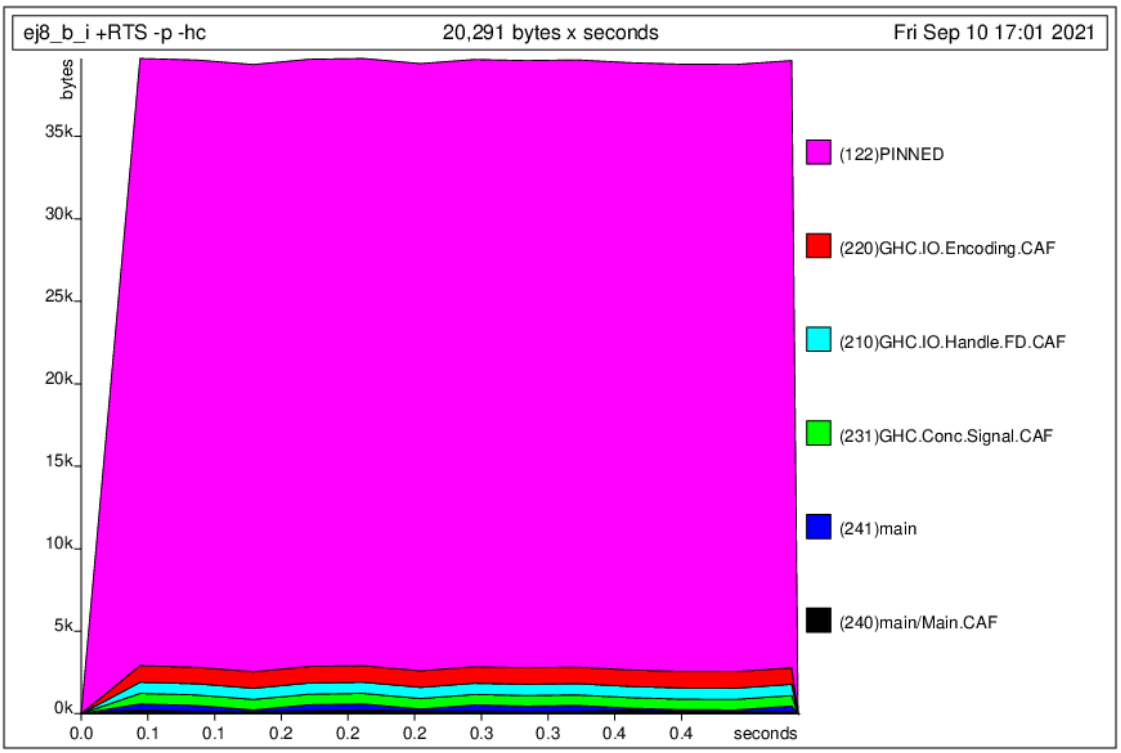
Se puede observar que esta versión es extremadamente lenta comparada a la anterior, pero tiene un menor consumo de memoria.

[3] GHC repository, Pinned Objects
[4] Stack Overflow, Heap is full of PINNED

8. (b) `main = print $ f [1..N] [1..N]`

8. (b) i. `f = \xs ys -> foldr (:) ys xs`

$N = 1000000$



En este caso `f` realiza la concatenación de dos listas y la memoria que necesita es tan poca, que el total de memoria *PINNED* resulta enorme en comparación y proviene probablemente del *runtime* del sistema en sí.[5]

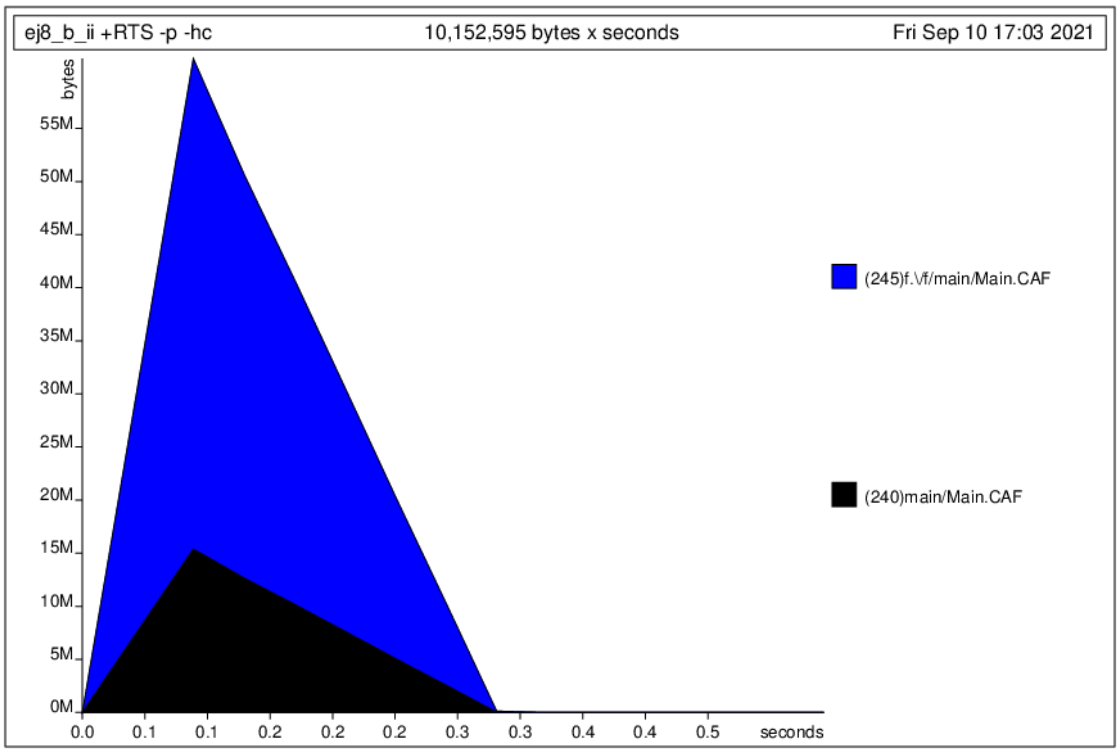
```
1 -- Primer Ejecución de foldr
2 (1 : (foldr (:) [1 .. n] [2 .. n]))
3 -- Segunda Ejecución de foldr (1 ya fue consumido debido a lazy evaluation)
4 (2 : (foldr (:) [1 .. n] [3 .. n]))
```

Debido a esta forma de ejecución es que se da esa meseta en el gráfico, dado que la memoria se va liberando a medida que se va generando.

[5] Stack Overflow, Heap is full of PINNED

8. (b) ii. `f = foldl (\k x -> k . (x:)) id`

$N = 1000000$



La ejecución de `f` resulta en una estructura de la siguiente forma:

```
-- Primer Ejecución de foldl
((id . (1:)) (foldl (\k x -> k . (x:)) id [2 .. n])) [1 .. n]
-- Segunda Ejecución de foldl
(((1:) . (2:)) (foldl (\k x -> k . (x:)) id [3 .. n])) [1 .. n]
-- Última Ejecución
((...((1:) . (2:)) . ... ) . (n:)) [1 .. n]
```

Por lo tanto, debido a que (:) necesita de una lista, se debe instanciar la lista completa antes de comenzar a imprimir.

Se da un caso parecido al (a).i, donde se notan las pendientes de generación y consumo de datos.

8. (c) `main = print f`

En ambos casos se utiliza $N = 10000000$.

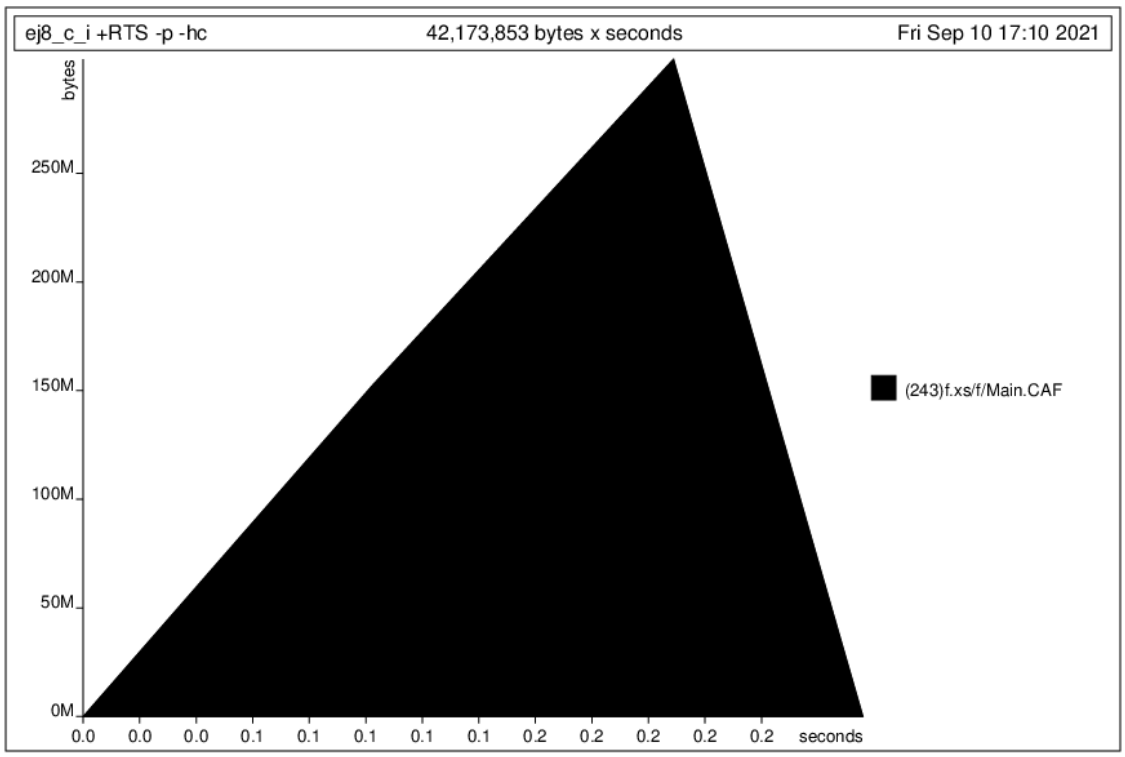


Figure 1: `f = let xs = [1..n] in if length xs > 0 then head xs else 0`

Aunque las dos funciones son prácticamente iguales y tienen el mismo tiempo de ejecución, el hecho de utilizar let para definir la variable xs implica que el mismo objeto será utilizado en la condición `length xs > 0` y al ejecutar el código `head xs`, por lo tanto, haskell no puede consumir y descartar xs al ejecutar la condición, lo cual sí ocurre al definirla estáticamente como en el otro caso, esto permite ir consumiendo y descartando la lista `[1..n]` dado que al ejecutar `head [1..n]` se utiliza otra lista igual, por lo que el consumo de memoria es ínfimo en comparación.

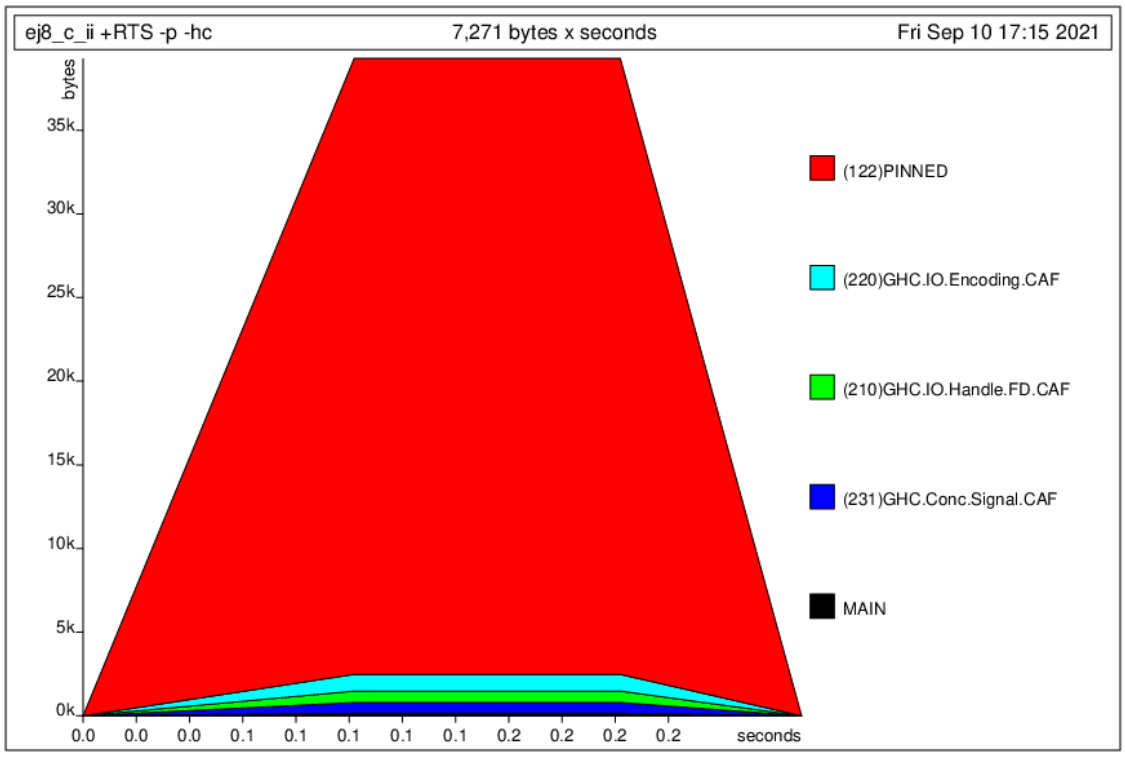


Figure 2: `f = if length [1..n] > 0 then head [1..n] else 0`