PRÁCTICA 1B

Aitor Melero Picón, Arturo Morcillo Penares

UAM Ampliación de Programación

PRÁCTICA 1b:

a) Implementar el problema de referencia inicial para el tipo "Float" sobre el precio y para el tipo "Int" sobre el código y la cantidad, y representando las ventas y los artículos como tuplas. Las operaciones con 2 argumentos deben ser operadores en lugar de funciones.

Hemos definido los tipos para los artículos, las ventas y las facturas de la siguiente manera:

```
-- Definicion de tipos
type Cantidad = Int
data Articulo = Articulo {codigo::Int, nombre::[Char], precio::Float}
type Venta = (Articulo, Cantidad)
type Factura = [Venta]
```

En primer lugar, tenemos una definición de tipo sinónimo para "Cantidad", realmente no es necesaria pero ayuda a ver la definición del tipo "Venta". Luego podemos ver la definición de un tipo algebraico "Articulo". Un "Articulo" está formado por una tupla con un valor de tipo "Int" para el código de un artículo, un valor de tipo "[Char]" (o "String", por eso el warning) para el nombre del artículo y un valor de tipo "Float" para el precio de un artículo. Debajo vemos la definición de un tipo sinónimo para un tipo "Venta" que es una tupla formada por artículos y las cantidades de dichos artículos. Finalmente, tenemos una asignación de un tipo sinónimo para el tipo "Factura" que será igual a una lista de ventas.

A continuación, iremos explicando una a una las funciones y operadores implementados para el presente apartado.

Primero tenemos la función "precioVenta":

```
-- Funcion que devuelve el precio de una venta
precioVenta :: Venta -> Float
precioVenta (a,c) = precio a * fromIntegral c
```

Esta función devuelve el precio de una venta, por lo tanto, tendremos un argumento de tipo "Venta" y una salida de un tipo "Float" ya que este es el tipo definido para el precio de una venta. Lo único que hay que destacar de esta función es que hemos necesita usar la función "fromIntegral" para pasar el tipo "Int" que representa la cantidad de un artículo de una venta a un tipo "Float" para poder calcular el precio total de la venta.

La siguiente función es "precioFactura":

```
-- Funcion que devuelve el precio de una factura
precioFactura :: Factura -> Float
precioFactura [] = 0
precioFactura (f:fs) = precioVenta f + precioFactura fs
```

Esta función calcula el precio de total de una factura. Debido a que para este apartado no podíamos usar funciones de orden superior, al igual que en otras funciones de este apartado, hemos tenido que hacer uso de la recursividad. Esta función va calculando el precio de cada venta de una factura y va sumando dichos precios de manera recursiva, como se acaba de mencionar.

Lo siguiente es el operador "($+$)":

```
-- Operador para la fusion de facturas
($+$) :: Factura -> Factura -> Factura
f1 $+$ f2 = f1 ++ f2
```

Este operador simplemente junta dos facturas formando una sola factura.

Lo siguiente es la función "sacarArticuloVenta":

```
-- Saca el articulo de una venta, necesaria por no tener deriving
sacarArticuloVenta :: Venta -> [Char]
sacarArticuloVenta (a,_) = nombre a
```

Esta función saca el nombre del artículo de una venta y hay que resaltar que es el nombre, no el artículo como tipo. Esa función se ha implementado más adelante porque aquí nos interesaba más el nombre del artículo, la función que saca el tipo "Artículo" no es compleja de implementar.

Luego tenemos el operador "($#)":

```
-- Operador para sacar el precio total de un articulo
($#) :: Factura -> Articulo -> Float
[] $# _ = 0
(v:vs) $# a
            | sacarArticuloVenta v == nombre a = precioVenta v + (vs $# a)
            | otherwise = vs $# a
```

Este operador saca el precio total de una factura. Para ello, va sumando el precio de cada venta de la factura hasta obtener el total.

Después tenemos la función "articuloACadena":

```
-- Funcion que devuelve el articulo en formato de cadena
articuloACadena :: Articulo -> String
articuloACadena a = "Articulo " ++ nombre a ++ " con codigo " ++ show (codigo a) ++ " y precio: " ++ show (precio a)
```

Esta función simplemente pasa un tipo "Articulo" a un tipo "[Char]", es decir, un tipo que muestra los datos de un artículo por pantalla en formato de cadena.

La siguiente función es "facturaACadena":

```
-- Funcion que devuelve una factura en formato de cadena
facturaACadena :: Factura -> String
facturaACadena [] = "Factura vacia"
facturaACadena [v] = ventaACadena v
facturaACadena (v:vs) = ventaACadena v ++ " || " ++ facturaACadena vs
```

Esta función hace lo mismo que la función anterior pero para un tipo "Factura".

Después tenemos el operador "(!#)":

```
-- Operador para sacar las ventas de un articulo en una factura
(!#) :: Factura -> Articulo -> Int
[] !# a = 0
(v:vs) !# a
    | sacarArticuloVenta v == nombre a = cantidadVenta v + (vs !# a)
    | otherwise = vs !# a
    where
        cantidadVenta (a, c) = c
```

Este operador la cantidad de un determinado articulo en una factura, se hace de manera recursiva también. Cabe destacar que para compara artículos aquí usamos el nombre del propio artículo, pero como hemos hecho más adelante, se puede hacer con el código en lugar del con el nombre.

Luego tenemos el operador "(!##)":

```
-- Operador para sacar las ventas de los articulos de una factura
(!##) :: Factura -> [Articulo] -> Int
[] !## a = 0
f !## [a] = f !# a
f !## (a:as) = (f !# a) + (f !## as)
```

Este operador hace lo mismo que el operador anterior, pero en lugar de para un solo artículo, calcula las ventas totales para varios artículos en una factura.

El siguiente es el operador "(-##)":

```
-- Operador que elimina articulos de una factura en funcion del articulo
(-##) :: Factura -> Articulo -> Factura
[] -## a = []
(v:vs) -## a
    | sacarArticuloVenta v == nombre a = vs -## a
    | otherwise = v : vs -## a
```

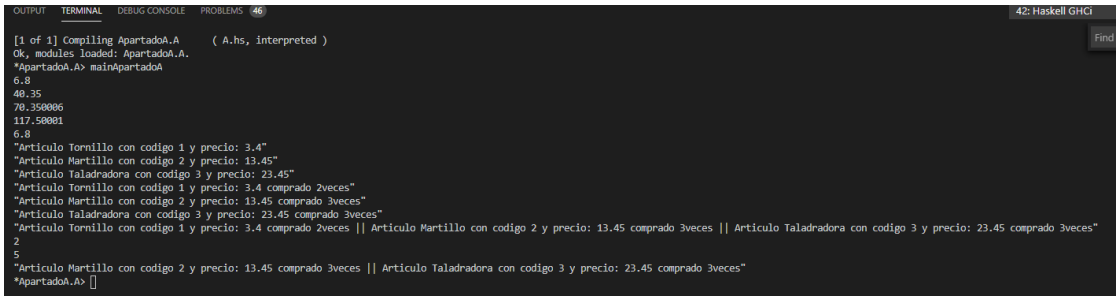
Este operador elimina un determinado artículo de una factura.

Por último, tenemos el operador "(-%)":

```
-- Operador que elimina articulo de una factura en relacion a la cantidad
(-%) :: Factura -> Int -> Factura
[] -% n = []
(v:vs) -% n
    | (v:vs) !# a < n = ((v:vs) -## a) -% n
    | otherwise = v : vs -% n
    where a = sacarArticulo v
          sacarArticulo (a,x) = a
```

Este operador elimina los artículos que estén por debajo de una cantidad determinada y devuelve la factura nueva.

Con respecto a los resultados, en el código se puede ver que en la función principal hemos creado diferentes artículos, ventas y facturas y hemos ido probando cada función. Estos han sido los resultados, aunque para entenderlos es mejor echar un vistazo a la función principal:



b) Repetir el apartado anterior para que se permita representar las ventas, por ejemplo, mediante: {VentaUnitaria microondas, Venta pan 3, etc.}, significando que hemos comprado 1 microondas y 3 barras de pan, y usando también data para el tipo de los artículos y de las facturas. Definir también el operador constructor (:+) para crear ventas. Hacer que los tipos de las facturas, las ventas y los artículos sean instancias de la clase Show.

Tal y como nos pide el enunciado renombramos las Ventas y las Facturas:

```
data Articulo = Articulo {codigo::Int, nombre::String, precio::Float}
data Venta = VentaUnitaria Articulo | VentaMultiple Articulo Cantidad | Articulo :+ Cantidad
data Factura = FacturaVacia | FacturaUnitaria Venta | FacturaMultiple [Venta]
```

De modo que ahora tenemos las ventas como ventas unitarias, ventas múltiples y con el operador constructor. Del mismo modo, también hemos redefinido las facturas.

De momento no podemos usar el operador deriving por lo que tenemos que reescribir la instancia de show para Articulo:

```
instance Show Articulo where
    show a = "Articulo " ++ nombre a ++ " con codigo " ++ show (codigo a) ++ " y precio: " ++ sh
ow (precio a)
```

Para venta:

```
instance Show Venta where
    show (VentaUnitaria a) = "Venta => Articulo " ++ nombre a ++ " con codigo " ++ show (codigo a) ++ " y precio: " ++ show (pre
cio a)
    show (VentaMultiple a c) = "Venta => Articulo " ++ nombre a ++ " con codigo " ++ show (codigo a) ++ " y precio: " ++ show (p
recio a) ++ " comprado " ++ show c ++ " veces"
    show (a :+ 1) = "Venta => Articulo " ++ nombre a ++ " con codigo " ++ show (codigo a) ++ " y precio: " ++ show (precio a)
    show (a :+ c) = "Venta => Articulo " ++ nombre a ++ " con codigo " ++ show (codigo a) ++ " y precio: " ++ show (precio a) ++
 " comprado " ++ show c ++ " veces"
```

Y para factura:

```haskell
instance Show Factura where
    show FacturaVacia = "Factura Vacia"
    show (FacturaUnitaria v) = "Factura con una sola venta: " ++ show v
    show (FacturaMultiple [v]) = show v
    show (FacturaMultiple (v:vs)) = show v ++ " || " ++ show (FacturaMultiple vs)
```

Como solución alternativa a la instancia de Venta podemos mostrar directamente el articulo con el "show" de articulo (Hecho en el apartado H).

Eso es lo principal. Una vez hecho eso fue necesario rehacer todas las funciones para asegurarnos que funciona sin problema. Para mantener la memoria sin demasiado código mostraremos el ejemplo de "$+$" y lo explicaremos.

```haskell
($+$) :: Factura -> Factura -> Factura
(FacturaUnitaria v1) $+$ FacturaVacia = FacturaUnitaria v1
FacturaVacia $+$ (FacturaUnitaria v2) = FacturaUnitaria v2
(FacturaMultiple v1) $+$ FacturaVacia = FacturaMultiple v1
FacturaVacia $+$ (FacturaMultiple v2) = FacturaMultiple v2
(FacturaUnitaria v1) $+$ (FacturaUnitaria v2) = FacturaMultiple [v1, v2]
(FacturaUnitaria v1) $+$ (FacturaMultiple v2) = FacturaMultiple (v1:v2)
(FacturaMultiple v1) $+$ (FacturaUnitaria v2) = FacturaMultiple (v2:v1)
(FacturaMultiple v1) $+$ (FacturaMultiple v2) = FacturaMultiple (v1 ++ v2)
```

Viendo esta función podemos ver que si queremos unir dos facturas tenemos que considerar toda la casuística. Al tener 3 tipos de facturas y esto usar dos facturas tenemos en total 8 casos. De no hacerlo así nos dirá que la definición es incompleta.

Por lo tanto, para redefinir las funciones se hace necesario mencionar cada posibilidad en las definiciones

c) Usando listas por comprensión y funciones de orden superior, mostrar cómo se podría codificar alternativamente y de un modo más compacto todo lo implementado en el apartado anterior. Además, añadir la operación de eliminar repeticiones de artículos en una factura.

Partiendo del código del apartado anterior, hemos realizado diferentes cambios para no tener que usar recursividad en muchas de las funciones implementadas.

El primer cambio es el siguiente:

```haskell
precioFactura :: Factura -> Float
precioFactura FacturaVacia = 0
precioFactura (FacturaUnitaria v) = precioVenta v
precioFactura (FacturaMultiple [v]) = precioVenta v
--CAMBIO 1
precioFactura (FacturaMultiple (v:vs)) = foldr (+) 0 [precioVenta p | p <- v:vs]
```

En este cambio, en lugar de llamarse a sí misma la función después, hemos creado una lista por comprensión que, por cada elemento de la lista de ventas, es decir, por cada venta de una factura, sacamos su precio. Después, usando "foldr" aplicamos el operador "(+)" con asociatividad por la derecha a la lista por comprensión. De esta manera sumamos el precio de cada venta de una factura obteniendo así el precio total de la factura.

El segundo cambio es:

```haskell
-- precioTotalArticulo ($#)
($#) :: Factura -> Articulo -> Float
(FacturaUnitaria v) $# a
            | sacarArticuloVenta v == nombre a = precioVenta v
            | otherwise = 0
(FacturaMultiple [v]) $# a
            | sacarArticuloVenta v == nombre a = precioVenta v
            | otherwise = 0
-- CAMBIO 2
(FacturaMultiple (v:vs)) $# a = foldr (+) 0 [precioVenta p | p <- v:vs, sacarArticuloVenta p == nombre a]
```

De manera similar a la función anterior, generamos una lista formada por los precios de aquellos artículos seleccionados (haciendo una comparación por el nombre de artículo, se puede hacer por código también) y se van sumando uno a uno hasta obtener el precio total para dicho artículo. Ese precio es lo que devuelve el operador.

El siguiente cambio es:

```haskell
-- ventasArticulo (!#)
(!#) :: Factura -> Articulo -> Int
FacturaVacia !# _ = 0
(FacturaUnitaria v) !# a
            | sacarArticuloVenta v == nombre a = 1
            | otherwise = 0
(FacturaMultiple [v]) !# a
            | sacarArticuloVenta v == nombre a = cantidadVenta v
            | otherwise = 0
            where
                  cantidadVenta (VentaUnitaria _) = 1
                  cantidadVenta (VentaMultiple _ c) = c
                  cantidadVenta (_ :+ c) = c
-- CAMBIO 3
(FacturaMultiple (v:vs)) !# a = cantidadVenta [v1 | v1 <- v:vs, sacarArticuloVenta v1 == nombre a]
                              where
                                    cantidadVenta [VentaUnitaria _] = 1
                                    cantidadVenta [VentaMultiple _ c] = c
                                    cantidadVenta [_ :+ c] = c
```

De nuevo, como en los casos anteriores, en lugar de usar recursividad hemos usado una lista por comprensión formada por las cantidades para un artículo determinado en una determinada factura. Vamos sumando dichas cantidades y se obtiene la cantidad total de artículos vendidos.

Luego tenemos el siguiente cambio:

```haskell
-- ventasArticulos (!##)
(!##) :: Factura -> [Articulo] -> Int
f !## [a] = f !# a
-- CAMBIO 4
f !## (a:as) = foldr (+) 0 [f !# a' | a' <- a:as]
```

Vamos sumando las cantidades de un artículo determinado para cada venta de una factura obteniendo así el número de artículos vendidos de ese artículo determinado en una factura.

El quinto cambio es:

```haskell
-- eliminacionArticulos (-##)
(-##) :: Factura -> Articulo -> Factura
FacturaVacia -## _ = FacturaVacia
(FacturaUnitaria v) -## a
    | sacarArticuloVenta v == nombre a = FacturaVacia
    | otherwise = FacturaUnitaria v
(FacturaMultiple [v]) -## a
    | sacarArticuloVenta v == nombre a = FacturaVacia
    | otherwise = FacturaUnitaria v
-- CAMBIO 5
(FacturaMultiple (v:vs)) -## a = f'' f'
                        where f' = [ f | f <- v:vs, sacarArticuloVenta f /= nombre a]
                              f'' [] = FacturaVacia
                              f'' [x] = FacturaUnitaria x
                              f'' (x:xs) = FacturaMultiple (x:xs)
```

Usando la misma idea, primero generamos una lista por comprensión formada por los artículos que no son del tipo del artículo seleccionado. Después devolvemos una factura formada por dichas ventas de la lista de comprensión.

El último cambio realizado ha sido el siguiente:

```haskell
-- eliminacionCantida d (-%)
(-%) :: Factura -> Int -> Factura
FacturaVacia -% _ = FacturaVacia
(FacturaUnitaria v) -% n
    | cantidadVenta v < n = FacturaVacia
    | otherwise = FacturaUnitaria v
    where
          cantidadVenta (VentaUnitaria _) = 1
          cantidadVenta (VentaMultiple _ c) = c
          cantidadVenta (_ :+ c) = c
(FacturaMultiple [v]) -% n
    | cantidadVenta v < n = FacturaVacia
    | otherwise = FacturaUnitaria v
    where
          cantidadVenta (VentaUnitaria _) = 1
          cantidadVenta (VentaMultiple _ c) = c
          cantidadVenta (_ :+ c) = c
-- CAMBIO 6
(FacturaMultiple (v:vs)) -% n = f'' f'
                       where f' = [ f | f <- v:vs, cantidadVenta f >= n]
                             cantidadVenta (VentaUnitaria _) = 1
                             cantidadVenta (VentaMultiple _ c) = c
                             cantidadVenta (_ :+ c) = c
                             f'' [] = FacturaVacia
                             f'' [x] = FacturaUnitaria x
                             f'' (x:xs) = FacturaMultiple (x:xs)
```

De nuevo podemos ver, y de manera similar al anterior operador, que hemos creado una lista por comprensión formada por las ventas que están por encima de un determinado número de artículos vendidos. Finalmente devolvemos la factura formada por las ventas de la lista de comprensión.

Además de realizar los cambios mencionados en el código del apartado anterior, también se pide implementar una función nueva que agrupase las ventas con artículos del mismo tipo de una factura teniendo las mismas cantidades para dichos artículos.

La idea (algoritmo) para esta función está formada por los siguientes pasos:

1. Ordenamos en orden ascendente todas las ventas de la factura por código de artículo.
2. Agrupamos las ventas de la lista ordenada del punto anterior por código de artículo.
3. Sumamos las cantidades de las ventas agrupadas.
4. Creamos la factura formada por las ventas a las que se llega por los 3 puntos anteriores.

El código de la función es:

```
-- Funcion extra del apartado C
eliminacionArticulosRepetidos :: Factura -> Factura
eliminacionArticulosRepetidos (FacturaMultiple (v:vs)) = FacturaMultiple (map sumarCantidad (agrupar (v:vs)))
                        where sumarCantidad :: [Venta] -> Venta
                              sumarCantidad (v':vs') =  VentaMultiple (articulo v') (sum [cantidad x | x <- v':vs'])
                              agrupar :: [Venta] -> [[Venta]]
                              agrupar (z:zs) = groupBy esCodigoIgual (ordenar (z:zs))
                              esCodigoIgual :: Venta -> Venta ->Bool
                              esCodigoIgual v1 v2 = codigo (articulo v1) == codigo (articulo v2)
                              ordenar :: [Venta] -> [Venta]
                              ordenar [] = []
                              ordenar (x:xs) = ordenar [ y | y <- xs, codigo (articulo (head xs)) <= codigo (articulo x)] ++ [x]
```

En el código se puede ver como se han ido realizando los pasos del algoritmo desde la parte inferior a la parte superior. La función “ordenar” devuelve una lista de ventas ordenadas por código de artículo. La función “esCodigoIgual” comprueba si los artículos de dos ventas son el mismo artículo (y esta vez comparamos el código y no el nombre del artículo). La función “agrupar” agrupa los elementos de la lista por comprensión ordenada por código de artículo, para ello usamos la función “groupBy”. Finalmente, usando la función “map” vamos sumando las cantidades de las ventas agrupadas de la lista generada en la función anterior obteniendo así una lista de ventas. Esa lista final de ventas es la factura final.

Para el c, hemos generado la misma salida que en los apartados anteriores, pero el resultado que nos interesa es el siguiente:

```
"Venta => Articulo Tornillo con codigo 1 y precio: 3.4 || Venta => Articulo Martillo con codigo 2 y precio: 13.45 comprado 2 veces || Venta => Articulo Taladradora con codigo 3 y precio: 23.45 comprado 3 veces || Venta => Articulo Martillo con codigo 2 y precio:
 13.45 comprado 3 veces"
Venta => Articulo Tornillo con codigo 1 y precio: 3.4 comprado 1 veces || Venta => Articulo Martillo con codigo 2 y precio: 13.45 comprado 5 veces || Venta => Articulo Taladradora con codigo 3 y precio: 23.45 comprado 3 veces
```

Como se puede apreciar, tenemos una factura con ventas con el mismo artículo. Después de aplicar la función anterior, podemos ver que ahora no aparecen ventas con artículos repetidos y que para las ventas con los artículos repetidos de la factura anterior se han sumado las cantidades de dichos artículos. Ahora no hay dos ventas con el artículo con código 2, ahora hay una venta con el artículo con código 2 y las cantidades totales.

d) Hacer tratamiento de errores a partir del apartado anterior, para casos como cantidades negativas o nombres/precios diferentes de artículos para el mismo código, invocando a la función error o usando el tipo Maybe. Incluir también la operación de simplificación de facturas en las que existan ventas relativas al mismo artículo, agrupando por artículo.

A la hora de realizar este ejercicio copiamos todo el código del apartado C y creamos funciones que comprueban la validez de los artículos:

```
-- Funciones para realizar la comprobacion de los articulos
checkArticulo :: Articulo -> Bool
checkArticulo a
    | null (nombre a) = error "Articulo sin nombre"
    | codigo a <= 0 = error "Articulo con codigo negativo o '0'"
    | precio a <= 0 = error "Articulo con precio nulo o negativo"
    | otherwise = True
```

De las ventas:

```
-- Funciones para realizar la comprobacion de las ventas
checkVenta :: Venta -> Bool
checkVenta (VentaUnitaria a) = checkArticulo a
checkVenta (VentaMultiple a c) = checkArticulo a && checkCantidad c
checkVenta (a :+ c) = checkArticulo a && checkCantidad c
```

De las facturas:

```
-- Funciones para realizar la comprobacion de las facturas
checkFactura :: Factura -> Bool
checkFactura FacturaVacia = True
checkFactura (FacturaUnitaria v) = checkVenta v
checkFactura (FacturaMultiple ventas) = and booleans
    where booleans = map checkVenta ventas
```

Y de las cantidades:

```
checkCantidad :: Cantidad -> Bool
checkCantidad c
    | c <= 0 = error "Cantidad negativa"
    | otherwise = True
```

Una vez hecho esto, simplemente comprobábamos en cada función que el parámetro era correcto antes de ejecutarla. Mostramos un ejemplo para mantener la memoria lo más resumida y compacta posible:

```
precioVenta :: Venta -> Float
precioVenta (VentaUnitaria a)
    | checkVenta (VentaUnitaria a) = precio a
precioVenta (VentaMultiple a c)
    | checkVenta (VentaMultiple a c) = precio a * fromIntegral c
precioVenta (a :+ 1)
    | checkVenta (a :+ 1) = precio a
precioVenta (a :+ c)
```

```
    | checkVenta (a :+ c) = precio a * fromIntegral c
```

e) Repetir el apartado c) considerando operaciones polimórficas válidas para tipos cualesquiera de la clase "Fractional" y de la clase "Integral" considerados para el precio, el código y la cantidad.

Basándonos en el código del apartado c) pero con los nuevos requisitos, tenemos la siguiente definición de tipos:

```
-- Modificamos los data para tipos polimorficos
data Articulo a b where
    Articulo :: (Integral a, Fractional b) => a -> String -> b -> Articulo a b

data Venta a b c where
    VentaUnitaria :: (Integral a , Fractional b) => Articulo a b -> Venta a a b
    VentaMultiple :: (Integral a, Fractional b) => Articulo a b -> a -> Venta a a b
    (:+) :: (Integral a, Fractional b) => Articulo a b -> a -> Venta a a b

data Factura a b c where
    FacturaVacia :: (Integral a, Fractional b) => Factura a a b
    FacturaUnitaria :: (Integral a, Fractional b) => Venta a a b -> Factura a a b
    FacturaMultiple :: (Integral a, Fractional b) => [Venta a a b] -> Factura a a b
```

Podemos ver que ahora un artículo se construye con un tipo polimórfico de la clase "Integral" que representa el código del artículo, un tipo "[Char]" para el nombre del artículo y un tipo polimórfico de la clase "Fractional" que representa el precio del artículo.

Para el tipo "Venta" y "Factura" partimos de la misma base, ahora se forman con tipos polimórficos.

En el resto del código hemos modificado todos los parámetros que de tipo "Int" y de tipo "Float" por los tipos polimórficos correspondientes. Lo único que nos ha complicado un poco más con respecto a este cambio ha sido la instancia de "Show", en ese caso hemos hecho que los tipos polimórficos correspondientes sean instancias de la clase "Show" también. Aquí se pueden ver dichos cambios realizados para poder imprimir por pantalla los artículos, las ventas y las facturas:

```
-- Instancias de Show
instance (Integral a, Show a, Fractional b, Show b) => Show (Articulo a b) where
    show a = "Articulo " ++ nombre a ++ " con codigo " ++ show (codigo a) ++ " y precio: " ++ show (precio a)

instance (Integral a, Show a, Fractional b, Show b) => Show (Venta a a b) where
    show (VentaUnitaria a) = "Venta => Articulo " ++ nombre a ++ " con codigo " ++ show (codigo a) ++ " y precio: " ++ show (pre
    show (VentaMultiple a c) = "Venta => Articulo " ++ nombre a ++ " con codigo " ++ show (codigo a) ++ " y precio: " ++ show (p
    show (a :+ 1) = "Venta => Articulo " ++ nombre a ++ " con codigo " ++ show (codigo a) ++ " y precio: " ++ show (precio a)
    show (a :+ c) = "Venta => Articulo " ++ nombre a ++ " con codigo " ++ show (codigo a) ++ " y precio: " ++ show (precio a) ++

instance (Integral a, Show a, Fractional b, Show b) => Show (Factura a a b) where
    show FacturaVacia = "Factura Vacia"
    show (FacturaUnitaria v) = "Factura con una sola venta: " ++ show v
    show (FacturaMultiple [v]) = show v
    show (FacturaMultiple (v:vs)) = show v ++ " || " ++ show (FacturaMultiple vs)
```

Con respecto a los resultados del apartado, nada que añadir, ya que hemos usado los mismos casos que en apartados anteriores para poder comprobar que todo funcionaba correctamente.

f) A partir del apartado anterior, permitir que el precio de una venta y de factura se calculen mediante 2 funciones con el mismo nombre (precio), haciendo que los tipos correspondientes pertenezcan a una clase conteniendo el método precio.

Hay que destacar que hemos copiado el código del apartado anterior y dentro del mismo hemos realizado los cambios pedidos. Preferimos hacer esto a crear otro fichero más para tener el menor número posible de ficheros. Los cambios del apartado f) se ven bien porque los hemos indicado, esos cambios son los siguientes:

```
-- Apartado F
-- Clase con el metodo precio

class Preciable (p :: * -> * ) where
    precio :: (Fractional b) => p b -> b

-- Instancias de precio
instance (Integral a) => Preciable (Venta a a) where
    precio = precioVenta

instance (Integral a) => Preciable (Factura a a) where
    precio = precioFactura
```

Hemos creado una clase “Preciable” para un tipo “Kind”, es decir, una especie de función. La clase está formada por el método “precio”. Este método recibe una función como argumento que recibe un argumento y devuelve un tipo “Fractional” que es el tipo del precio de un artículo.

Instanciamos primero un tipo “Venta” para nuestra clase “Preciable”. Ese tipo “Venta”, basándonos en nuestra definición de tipo “Venta” del apartado anterior, tendrá el tipo polimórfico “a” de la clase “Integral”. Entonces, “precio” simplemente se igualará a nuestra función para calcular el precio de una venta llamada “precioVenta”. Explicándonos mejor, la función “precio” devuelve una función a la que le metes un tipo “Venta” y te devuelve su precio y es por ello que “precio” se puede asignar directamente a “precioVenta”.

Hacemos lo mismo con un tipo “Factura” y esto nos permite poder usar la función “precio” con un tipo “Venta” o “Factura” independientemente porque, ya definida la función para cada instanciación, se llamará a la función “precioVenta” o “precioFactura” dependiendo del tipo.

Después de cambiar todas las llamadas a “precio” hemos visto que los resultados son los mismos que los de apartados anteriores para las mismas operaciones, lo que nos indica que el cambio es correcto.

g) Repetir el apartado c) considerando los elementos de una factura distribuidos en un árbol, en lugar de una lista, con una estructura semejante a: “data Arbol = Arb a [Arbol]”.

La forma de redefinir la factura fue la siguiente:

```
data Factura = FacturaVacia | FacturaUnitaria Venta | FacturaMultiple Venta [Factura]
```

Donde tenemos la FacturaVacia y FacturaUnitaria como hojas. En la facturaMultiple pueden haber más ramas.

Las estrategias para rehacer las funciones fueron dos. Mediante recursividad y rescribiendo el árbol.

Mediante recursividad simplemente llamamos a la función de nuevo para el resto de los elementos:

```
-- precioTotalArticulo ($#)
($#) :: Factura -> Articulo -> Float
FacturaVacia $# a = 0

(FacturaUnitaria v) $# a
            | sacarArticuloVenta v == nombre a = precioVenta v
            | otherwise = 0
(FacturaMultiple v []) $# a
            | sacarArticuloVenta v == nombre a = precioVenta v
            | otherwise = 0
(FacturaMultiple v [f]) $# a
            | sacarArticuloVenta v == nombre a = precioVenta v + f $# a
            | otherwise = 0

-- CAMBIO 2
(FacturaMultiple v (f:fs)) $# a = foldr (+) 0 [precioVenta p | p <- getVentasFactura (FacturaMul
tiple v (f:fs)), sacarArticuloVenta p == nombre a]
```

El método de reescritura del árbol consistió en crear dos funciones:

Una para obtener una lista de ventas del árbol de la factura:

```
getVentasFactura :: Factura -> [Venta]
getVentasFactura FacturaVacia = []
getVentasFactura (FacturaUnitaria v) = [v]
getVentasFactura (FacturaMultiple v []) = [v]
getVentasFactura (FacturaMultiple v [f]) = v:getVentasFactura f
getVentasFactura (FacturaMultiple v (f:fs)) = [v] ++ getVentasFactura f ++ getVentasFacturaAux f
s
    where getVentasFacturaAux [] = []
          getVentasFacturaAux (f:fs) = getVentasFactura f ++ getVentasFacturaAux fs
```

Otra para formar un nuevo árbol mediante la lista de ventas:

```
construirArbol :: [Venta] -> Factura
construirArbol [] = FacturaVacia
construirArbol [v] = FacturaUnitaria v
construirArbol (v:vs) = FacturaMultiple v (hacerFacturas vs)
    where hacerFacturas [v] = [FacturaUnitaria v]
          hacerFacturas (v:vs) = FacturaUnitaria v : hacerFacturas vs
```

Entonces podíamos usar las anteriores funciones y a partir de ellas crear un nuevo árbol como en el siguiente ejemplo:

```
(FacturaMultiple v [f]) -
## a = construirArbol (eliminarArticulos (getVentasFactura (FacturaMultiple v [f])) a)
```

h) Considerar una casuística más amplia para los artículos y las ventas, implementando la funcionalidad descrita en el apartado c). Por ejemplo, podrían existir distintos tipos de artículos: alimentos, electrodomésticos, etc, y alguno de estos artículos podría tener

promociones como "paga 2 y llévate 3", o cada tipo de artículo podría tener un diferente porcentaje de IVA como impuestos, etc. Con relación a ventas, podrían existir ventas a domicilio, con un precio mayor que una venta normal, etc.

Este apartado fue notablemente más sencillo que el anterior.

Comenzamos creando las promociones, ofertas y tipos de artículos. Junto a esto, creamos nuevas funciones para obtener los valores asociados:

```
-- Tipos posibles de articulos
data TipoArticulo = ProductoBasico | ProductoImportante | ProductoLujo
instance Show TipoArticulo where
    show ProductoBasico = "Producto basico"
    show ProductoImportante = "Producto importante"
    show ProductoLujo = "Producto de lujo"

--Obtencion del IVA aplicado por el articulo
getIva :: TipoArticulo -> Float
getIva ProductoBasico = 0.04
getIva ProductoImportante = 0.1
getIva ProductoLujo = 0.21
-- Tipos de ventas
data TipoVenta = Normal | ADomicilio | Regalo | RegaloADomicilio
instance Show TipoVenta where
    show Normal = "Normal"
    show ADomicilio = "a domicilio"
    show Regalo = "Regalo"
    show RegaloADomicilio = "Regalo a domicilio"

-- Obtencion del extra de las nuevas ventas
getExtraVenta :: TipoVenta -> Float
getExtraVenta Normal = 0
getExtraVenta ADomicilio = 10
getExtraVenta Regalo = 5
getExtraVenta RegaloADomicilio = getExtraVenta ADomicilio + getExtraVenta Regalo
-- Promociones
data Promocion = Ninguna | DosPorTres | MitadPrecio deriving Eq
```

Una vez hecho esto hicimos que Articulo y Venta lo implementasen:

```
data Articulo = Articulo {codigo::Int, nombre::String, precio::Float, tipo::TipoArticulo, promoc
ion::Promocion}
data Venta = VentaUnitaria Articulo TipoVenta | VentaMultiple Articulo Cantidad TipoVenta | Arti
culo :+ Cantidad
```

Y finalmente modificamos las funciones de obtener el precio de articulo y venta:

```
precioArticulo :: Articulo -> Float
precioArticulo a = precio a * (1 + getIva (tipo a))

getTipoVenta :: Venta -> TipoVenta
getTipoVenta (VentaUnitaria _ tv) = tv
getTipoVenta (VentaMultiple _ _ tv) = tv
```

```
getTipoVenta (_ :+ 1) = Normal
getTipoVenta (_ :+ _) = Normal

getCantidadGratis :: Int -> Int
getCantidadGratis c = round (fromIntegral  c/3)

precioVenta :: Venta -> Float
precioVenta (VentaUnitaria a t) = precioArticulo a + getExtraVenta t
precioVenta (VentaMultiple a c t)
    | promocion a == DosPorTres = (precioArticulo a * fromIntegral (c - getCantidadGratis c)) +
getExtraVenta t
    | promocion a == MitadPrecio = ((precioArticulo a / 2) * fromIntegral c) + getExtraVenta t
    | otherwise = (precioArticulo a * fromIntegral c) + getExtraVenta t
precioVenta (a :+ 1) = precioArticulo a
precioVenta (a :+ c)
    | promocion a == DosPorTres = precioArticulo a * fromIntegral (c - getCantidadGratis c)
    | promocion a == MitadPrecio = (precioArticulo a / 2) * fromIntegral (c - getCantidadGratis
c)
    | otherwise = precioArticulo a * fromIntegral c
```

Existiendo la gran ventaja de que no es necesario modificar las funciones correspondientes a Factura si hemos realizado estos cambios de manera correcta.