Monadic Short Cut Deforestation

Cecilia Manzino

Licenciatura en Ciencias de la Computación

Facultad de Ciencias Exactas, Ingeniería y Agrimensura

Universidad Nacional de Rosario

ceciliam@fceia.unr.edu.ar

Programación modular en programación funcional

- En programación funcional los programas suelen construirse usando un estilo composicional.
- Un programa en este estilo es expresado como una secuencia de transformaciones sobre estructuras de datos.

  Por ejemplo:

  ```
  all p xs = and (map p xs)
  ```

- Se obtienen programas claros, pero no necesariamente eficientes.

- Una versión de `all` que no crea listas intermedias es la siguiente:

```
all p xs = h xs
  where h [] = True
        h (x:xs) = p x && h xs
```

Deforestación

- Una técnica de transformación de programas para la eliminación de las estructuras de datos que se crean en la composición de funciones es llamada deforestación (Wadler 1988).

- Actualmente existen varios métodos que extendieron el algoritmo de Wadler, probablemente el más famoso es “Short Cut Deforestation”.

- Una función que consume una lista puede expresarse usando la función `foldr`:

$$foldr\ (\oplus)\ z\ [x_1, x_2, \ldots, x_n]\ =\ x_1\ (\oplus\ x_2\ (\oplus(\ldots(x_n\ \oplus\ z)))$$

- Una función que produce una lista puede expresarse usando la función `build`:

$$build\ g\ =\ g\ (:)\ []$$

Regla foldr/build

```
g:: forall b . (a -> b -> b ) -> b -> b
 =>  foldr k z ( build g ) = g k z
```

donde

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f n [] = n
foldr f n (x:xs) = f x (foldr f n xs)

build :: (forall b. (a -> b -> b) -> b -> b) -> [a]
build g = g (:) []
```

```
map p xs = case xs of
             [] -> []
             (x:xs') -> p x : map p xs'

map' p xs = (\c n -> case xs of
                       [] -> n
                       (x:xs') -> c (p x) (map' p xs'))


map p xs = build (map' p xs)

and xs = foldr (&&) True xs

all p xs
  = and (map p xs)
  = foldr (&&) True (build (map' p xs))
  = map' p xs (&&) True
```

Regla destroy/unfoldr

```
g :: forall a. (a -> Maybe(b, a)) -> a -> c
         =>  destroy g (unfoldr f e) = g f e
```

donde

```
unfoldr :: (b -> Maybe (a, b)) -> b -> [a]
unfoldr f b = case f b of
                Nothing -> []
                Just (a, b') -> a: unfoldr f b'

destroy ::(forall a.(a -> Maybe (b, a))-> a-> c)-> [b]-> c
destroy g xs = g listpsi xs
    where listpsi [] = Nothing
          listpsi (x:xs) = Just (x, xs)
```

Mónadas

- Las mónadas se usan en programación funcional para estructurar programas que producen efectos tales como excepciones, entrada/salida, estados globales y otros.
- Una mónada es una tripleta `(m, return, >>=)`, donde:
  1. `m` es un constructor de tipo.
  2. `return: a -> m a` es una función polimórfica, y
  3. `(>>=): m a -> ( a -> m b ) -> m b` es un operador polimórfico llamado bind.
- Permiten distinguir entre valores y cómputos. Un objeto de tipo `m a` es interpretado como un cómputo que devuelve un valor de tipo `a`.
- Una función de tipo $a \rightarrow m\ b$ es llamada monádica.

Monadic Shortcut Deforestation

Regla foldr/buildm

```
g :: Monad m => forall b. (a -> b -> b) -> b -> m b
  =>  do { xs <- buildm g ; return (foldr k z xs)}
      = g k z
```

donde

```
buildm::Monad m => (forall b.(a -> b ->b) -> b -> m b)
                      -> m [a]
buildm g = g (:) []
```

- La regla `destroy/unfoldr` restringida a casos monádicos es la siguiente:

```
g :: forall a. (a -> Maybe(b, a)) -> a -> m c
   =>  destroy g (unfoldr f e) = g f e
```

Implementación en GHC (Glasgow Haskell Compiler)

- La idea básica del mecanismo de reglas de reescritura de GHC es permitir al programador expresar reglas que serán tenidas en cuenta por el compilador.

- Las reglas de reescrituras son expresadas usando la sintaxis de Haskell, y deben estar contenidas en una expresión de la forma: `{-# .. #-}`.

Regla mmap/foldr/buildm

```
{-# RULES "mmap/foldr/buildm"
    forall k z (g::Monad m => (forall b. (a -> b -> b)
                                          -> b -> m b)) .
        mmap (foldr k z) (buildm g) = g k z

#-}
```

donde

```
mmap :: Monad m => ( a -> b) -> (m a -> m b)
mmap f m = m >>= \a -> return (f a)
```

- La regla de reecritura para `destroy/unfoldr` se define como:

```
{-# RULES "destroy/unfoldr"
    forall k e (g :: forall a. (a -> Maybe(b, a))
                                   -> a -> c) .
        destroy g (unfoldr f e) = g f e
#-}
```

Funciones de librerías en términos de buildm, foldr, destroy y unfoldr.

- Usando buildm: `zipWithM`, `getLine`, `unfoldM`, `accumulate`, `mapM` y `hgetContents`.
- Usando `foldr`: `concatMap`, `product`, `sum`, `and`, `or`, `and`, `unlines`, `map`, `length`, `mapM`, `accumulate`.
- Usando `destroy`: `PutStr`, `foldM`, `accumulate`, `mapM`, `hPutStr`.
- Usando `unfoldr`: `init`, `iterate`, `replicate`, `takeWhile`, `zip`, `zipWith`,`enumFromTo`.

Caso de estudio

```
main = do { xs <- hGetContents h;
            hPutStr h (filter (/='\^M') xs) }
```

Se introduce la función de `mmap`.

```
main = do{ xs <- mmap (filter (/='\^M')) (hGetContents h);
           hPutStr h xs }
```

Se utilizan las definiciones de `filter` y `hGetContents` en términos de `foldr` y `buildm`.

```
main = do { xs <- mmap  (foldr (\x xs-> if (x /='\^M')
                                           then (x:xs)
                                           else xs) [])
                        (buildm hgetC);
            hPutStr h xs }

  where hgetC c n = do { eof <- hIsEOF h;
                         if eof then hClose h >> return n
                         else do { x <- hGetChar h;
                                   xs <- hgetC c n;
                                   return (c x xs)} }
```

Aplicando la regla `mmap\foldr\buildm` y algunas β-reducciones se obtiene:

```
main = do { xs <- hgetC;
            hPutStr h xs }

 where hgetC = do { eof <- hIsEOF h;
                    if eof then hClose h >> return []
                    else do { x <- hGetChar h;
                              xs <- hgetC;
                              return (if (x /='\^M')
                                      then (x:xs)
                                      else xs)} }
```

Programa	Disminución del tiempo de ejecución (%)	Disminución del uso de memoria (%)
1	31.5	50.3
2	22.1	20.0
3	11.2	20.0
4	36.0	32.5
5	1.4	4.8
6	13.5	4.8
7	20.6	4.8
8	6.7	20.0
9	7.9	0.8
10	45.0	97.2

Figura 1: Resultados de aplicar la regla mmap/foldr/buildm

Programa	Disminución del tiempo de ejecución (%)	Disminución del uso de memoria (%)
1	16.3	21.6
2	24.6	16.7
3	9.7	15.7
4	20.4	17.7
5	28.3	16.9
6	10.8	15.1
7	16.4	20.0
8	25.5	17.7
9	28.5	38.1
10	28.8	20.0

Figura 2: Resultados de aplicar la regla destroy/unfoldr

Generalización

Ley de lluvia ácida monádica

Sea c un tipo fijo,

$$g : \forall a \,.\, (F\ a \to a) \to c \to m\ a$$
$$\Rightarrow\ (m\ fold_F\ (f)) \circ (g\ in_F) = g\ f$$

Ley de lluvia ácida monádica (dual)

Sea c un tipo fijo,

$$g : \forall a \,.\, (a \to F\ a) \to a \to m\ c$$
$$\Rightarrow\ g\ f\ =\ (g\ out_F)\ \circ\ unfold_F\ (f)$$

Regla mmap/foldR/buildmR

```
    g :: Monad m => (forall b.(a -> [b] -> b) -> b -> m b)
        mmap (foldR k z) (buildmR g) = g k z
```

donde

```
data Rose a = Empty | Fork a [Rose a]


buildmR :: Monad m =>(forall b.(a -> [b] ->b) -> b -> m b)
                                -> m (Rose a)
buildmR g = g Fork Empty


foldR :: (a -> [b] -> b) -> b -> Rose a -> b
foldR k z Empty = z
foldR k z (Fork a xs) = k a (map (foldR k z) xs)
```

Posibles trabajos futuros

- Construcción de un algoritmo que derive `foldr`, `buildm`, `destroy` y `unfoldr` a partir de definiciones resursivas.
- Construcción de un algoritmo que aplique una transformación sintáctica a los programas para introducir la función `mmap`.
- Definición de una versión monádica de la regla `foldr/augment`, la cual generaliza `foldr/build`.
- Implemtentación de un algoritmo que elija entre diferentes definiciones de funciones en términos de `foldr`, `buildm`, `destroy` y `unfoldr`, de manera que la reglas `mmap/foldr/buildm` y `destroy/unfoldr` puedan coexistir en GHC.