Métodos de Programação I

Trabalho Prático nº 2

Miguel dos Santos Esteves - Nº 43202
Nuno Miguel Mendonça Coutinho Correia - Nº 43179
Tiago Coutinho Correia - nº43179

23 de Dezembro de 2006

Resumo

Este trabalho consiste numa pequena aplicação que efectua a conversao de programas do estilo *pointwise* para o estilo *point-free* e vice-versa.

Seguidamente apresentamos o código do trabalho explicando passo a passo o seu algoritmo

Conteúdo

Parte I
Inicialização

Neste parte limitamo-nos a fazer o import de algumas bibliotecas necessárias ao longo do programa e a implementar os tipos de dados fornecidos no enunciado.

```
module TP2 where

import Language.Haskell.Syntax
import Char
import List
import Control.Monad.State
```

```
data Def = Exp :=: Exp | Def:&:Def
               deriving (Eq,Show)


data Exp = Id | Exp:.:Exp
           | Bang | Const Exp | Unit | Cata Exp | In | Out
           | Nil | Cons  --RList
           | Fst | Snd | Exp:/\:Exp | Exp:*:Exp
           | Inl | Inr | Exp:\/:Exp | Exp:+:Exp
           | Cond Exp Exp Exp | TRUE | FALSE
           | Fun String | Num Int
           | Var String | Pair Exp Exp | Exp:@:Exp
           | RList Int
               deriving (Eq,Show)


--Var : serve para representar variaveis
--Pair: construir pares explicitamente
--Unit: ()
--(:@:) : para representar a aplicaçao de uma funçao a um argumento
--(:/\:) : para representar o split
--(:.:) : para representar a composiçao de funçoes
--(:\/:) : para representar o either
```

1 Testes e exemplos

```
assocr1 :: Def
assocr1 = ((Fun "assocr" ) :@: (Pair ( Pair ( Var "x") (Var "y"))(Var "z")))
          :=:
         (Pair (Var "x") (Pair(Var "y")(Var "z")))

exp1 = (Pair (Var "x") (Pair(Var "y")(Var "z")))
exp2 = ((Id :*: Inl) :\/: (Id :*: Inr))
exp3 = Id
exp4 = (((Fun "f" ):*:(Fun "g")):.:((Fun "h"):/\:(Fun "i")))
exp5 = Pair (Fst :@: (Fst :@: Var "xyz")) (Pair (Snd :@: (Fst :@: Var "xyz")) (Snd :@: Var "xyz"))
def6 = ((Id:.:Id):@:Var "x"):=:((Fun "f":.:Fun "g"):@:Var"y")
def7 = (Fun "condicao":@: (Var "x")) :=: Cond (Fun "teste" :@: Var "x") (Fun "resultado1" :@: Var "x") (Fun

assocr2 :: Def
assocr2 = (Fun "assocr" ) :=: (( Fst :.: Fst) :/\: (Snd :*: Id))


undistr1 :: Def
undistr1 = (Fun "undistr") :=: ((Id :*: Inl) :\/: (Id :*: Inr))


undistr2 :: Def
undistr2 = (((Fun "undistr") :@: (Inl:@:(Pair (Var "x" )(Var "y"))))
           :=:
            (Pair (Var "x")(Inl :@: (Var "y"))))
           :&:
```

```
           (((Fun "undistr") :@: (Inr :@: (Pair (Var "x")(Var "y"))))
           :=:
           (Pair (Var "x")(Inr :@: (Var "y"))))

def_succ = ((Fun "succ") :.: ((Id :.: (Fun "succ") ) :.: ((Fun "succ") :.: Id)) :=:
           (((Fun "succ") :.: (Fun "succ")) :.: (Fun "succ")))

distwo = (Fun "distwo" :@: (Inl :@: Var "x") :=: Pair (Inl :@: Unit) (Var "x")) :&:
         (Fun "distwo" :@: (Inr :@: Var "x") :=: Pair (Inr :@: Unit) (Var "x"))

coswap = (((Fun "coswap" :@: (Inl :@: Var "x")) :=: (Inr :@: Var "x") )
         :&:
         ((Fun "coswap" :@: (Inr :@: Var "x")) :=: (Inl :@: Var "x") ))

swap = ((Fun "swap":@: (Pair (Var "x") (Var "y") )) :=: (Pair (Var "y") (Var "x")) )
swap2 = (Fun "swap2") :=: ((Snd :/\: Fst):.:(Snd :/\: Fst))
simp_swap2 = reflexao_x.cancelamento_x.fusao_x $  swap2

coswap2 = Fun "coswap2" :=: ((Inr :\/: Inl) :.: (Inr :\/: Inl))

mikstr :: Def
mikstr =   (((Fun "mikstr") :@: (Var "x"))
           :=:
            (Pair (Var "x") (Id:@: (Var "x"))))
          :&:
           ((((Fun "mikstr") :@: (Inl:@:(Pair (Var "x" )(Var "y"))))
           :=:
            (Pair (Var "x")(Inl :@: (Var "y"))))
          :&:
           (((Fun "mikstr") :@: (Inr :@: (Pair (Var "x")(Var "y"))))
           :=:
           (Pair (Var "x")(Inr :@: (Var "y")))))


inb = (Fun "inb":@:(Inl:@:Unit) :=: TRUE) :&: (Fun "inb":@:(Inr:@:Unit) :=: FALSE)

condi = (Fun "condi" :@: Var "x") :=: (Cond (Fun "f":@: Var "x" ) TRUE FALSE)


def_len = ((Fun "len" :@: Nil) :=: Num 0)
          :&:
          ((Fun "len" :@: (Cons :@: (Pair (Var "h") (Var "t")))) :=: (Fun "succ" :@: (Fun "len" :@: Var "t"

--pwpf_len = invfusao_mais.igualdade_either.ext_igual.composicao.composicao.elim_x.invdef_const $ def_len

def_rev = ((Fun "rev" :@: Nil) :=: Nil)
          :&:
          ((Fun "rev" :@: (Cons :@: (Pair (Var "h") (Var "t")))) :=: (Fun "smoc" :@: (Pair (Var "h") (Fun "


def_soma = ((Fun "soma" :@: Nil) :=: Num 0)
          :&:
          ((Fun "soma" :@: (Cons :@: (Pair (Var "h") (Var "t")))) :=: (Fun "plus" :@: (Pair (Var "h") (Fun "


def_conta = ((Fun "conta" :@: Nil) :=: Num 0)
          :&:
          ((Fun "conta" :@: (Cons :@: (Pair (Var "h") (Var "t")))) :=: Cond (Fun "p" :@: Var "h") (Fun "succ
```

```
def_filter = ((Fun "filter p" :@: Nil) :=: Nil)
          :&:
          ((Fun "filter p" :@: (Cons :@: (Pair (Var "h") (Var "t")))) :=: Cond (Fun "p" :@: Var "h") (Cons
```

```
---- PROVAS ----

pwpf_assocr = invdef_x.invnatural_id.ext_igual.invdef_split.invdef_split.composicao.elim_x.elim_x $ assocr1
pfpw_assocr = invelim_x.invelim_x.invcomposicao.def_split.def_split.invext_igual.natural_id.def_x $ assocr2

pwpf_undistr = invuniversal_mais.invdef_x.invnatural_id.ext_igual.invdef_split.composicao.elim_x.composicao
pfpw_undistr = invcomposicao.invelim_x.invcomposicao.def_split.invext_igual.natural_id.def_x.universal_mais

pfpw_succ = def_id.def_id.invcomposicao.invcomposicao.invcomposicao.invcomposicao.invext_igual $ def_succ

pwpf_distwo = invdef_mais.troca.invuniversal_mais.ext_igual.invdef_split.composicao.invdef_bang $ distwo
pfpw_distwo = def_id.def_bang.invcomposicao.def_split.invext_igual.universal_mais.troca.def_mais $ pwpf_dist

pwpf_coswap = invuniversal_mais.ext_igual.composicao $ coswap
pfpw_coswap = invcomposicao.invext_igual.universal_mais $ pwpf_coswap

pwpf_swap = ext_igual.invdef_split.elim_x $ swap
pfpw_swap = invelim_x.def_split.invext_igual $ pwpf_swap

pwpf_inb = invuniversal_mais.ext_igual.composicao.invdef_const $ inb
pfpw_inb = def_const.invcomposicao.invext_igual.universal_mais $ pwpf_inb
```

Parte II
Leis

2 PRODUTO

```
--PRODUTO--

def_split :: Def -> Def
def_split = penetra def_sp
        where def_sp ((f:/\:g):@:x) = Pair (f:@:x) (g:@:x)
              def_sp v = v

invdef_split :: Def -> Def
invdef_split = penetra invdef_sp
        where invdef_sp (Pair (f:@:x) (g:@:y)) | x==y = (f:/\:g):@:x
              invdef_sp (Pair (f:@:x) y) | x==y = (f:/\:Id):@:x
              invdef_sp (Pair x (g:@:y)) | x==y = (Id:/\:g):@:x
              invdef_sp (Pair x y) | x==y = (Id:/\:Id):@:x
              invdef_sp v = v

def_x :: Def -> Def
def_x = penetra df_x
```

```
        where df_x (f :*: g) = ((f:.:Fst):/\:(g:.:Snd))
              df_x v = v

invdef_x :: Def -> Def
invdef_x = penetra invdf_x
        where invdf_x ((f:.:Fst):/\:(g:.:Snd)) = (f:*:g)
              invdf_x v = v

universal_x :: Def -> Def
universal_x (k:=:(f:/\:g)) = ((Fst:.:k):=:f) :&: ((Snd:.:k):=:g)
universal_x ((k:@:q):=:((f:/\:g):@:w)) | q==w = (((Fst:.:k):@:q):=:(f:@:q)) :&: (((Snd:.:k):@:q):=:(g:@:q))
universal_x (z:&:x) = (universal_x z) :&: (universal_x x)
universal_x z = z

invuniversal_x :: Def -> Def
invuniversal_x z@(((Fst:.:k):=:f) :&: ((Snd:.:w):=:g)) | k==w = (k:=:(f:/\:g))
invuniversal_x (z:&:x) = (invuniversal_x z) :&: (invuniversal_x x)
invuniversal_x z = z


cancelamento_x :: Def -> Def
cancelamento_x = penetra canc_x
        where canc_x (Fst:.:(f:/\:g)) = f
              canc_x (Snd:.:(f:/\:g)) = g
              canc_x v = v

reflexao_x :: Def -> Def
reflexao_x = penetra ref_x
        where ref_x (Fst:/\:Snd) = Id
              ref_x v = v

fusao_x :: Def -> Def
fusao_x = penetra fus_x
        where fus_x ((g:/\:h):.:f) = ((g:.:f):/\:(h:.:f))
              fus_x v = v

invfusao_x :: Def -> Def
invfusao_x = penetra invfus_x
        where invfus_x ((g:.:f):/\:(h:.:f2)) | f==f2 = ((g:/\:h):.:f)
              invfus_x v = v


absorcao_x :: Def -> Def
absorcao_x = penetra absor_x
        where absor_x ((i:*:j):.:(g:/\:h)) = ((i:.:g):/\:(j:.:h))
              absor_x v = v

functor_x :: Def -> Def
functor_x = penetra funct_x
        where funct_x ((g:.:h):*:(i:.:j)) = ((g:*:i):.:(h:*:j))
              funct_x v = v

functor_id_x :: Def -> Def
functor_id_x = penetra funct_id_x
        where funct_id_x (Id:*:Id) = Id
              funct_id_x v = v
```

3 COPRODUTO

```
--COPRODUTO--


def_mais :: Def -> Def
def_mais = penetra d_mais
        where d_mais (f:+:g) = ((Inl:.:f):\/:(Inr:.:g))
              d_mais v = v

invdef_mais :: Def -> Def
invdef_mais = penetra invd_mais
        where invd_mais ((Inl:.:f):\/:(Inr:.:g)) = (f:+:g)
              invd_mais v = v


universal_mais :: Def -> Def
universal_mais (k:=:(f:\/:g)) = ((k:.:Inl):=:f) :&: ((k:.:Inr):=:g)
universal_mais ((k:@:q):=:((f:\/:g):@:w)) | q==w = (((k:.:Inl):@:q):=:(f:@:q)) :&: (((k:.:Inr):@:q):=:(g:@:
universal_mais (z:&:x) = (universal_mais z) :&: (universal_mais x)
universal_mais z = z


invuniversal_mais :: Def -> Def
invuniversal_mais z@(((k:.:Inl):=:f) :&: ((w:.:Inr):=:g)) | k==w = (k:=:(f:\/:g))
invuniversal_mais (z:&:x) = (invuniversal_mais z) :&: (invuniversal_mais x)
invuniversal_mais z = z



cancelamento_mais :: Def -> Def
cancelamento_mais = penetra canc_mais
        where canc_mais ((f:\/:g):.:Inl) = f
              canc_mais ((f:\/:g):.:Inr) = g
              canc_mais v = v

reflexao_mais :: Def -> Def
reflexao_mais = penetra ref_mais
        where ref_mais (Inl:\/:Inr) = Id
              ref_mais v = v

fusao_mais :: Def -> Def
fusao_mais = penetra fus_mais
        where fus_mais (f:.:(g:\/:h)) = ((f:.:g):\/:(f:.:h))
              fus_mais v = v

invfusao_mais :: Def -> Def
invfusao_mais = penetra invfus_mais
        where invfus_mais ((f1:.:g):\/:(f2:.:h)) | f1==f2 = (f1:.:(g:\/:h))
              invfus_mais v = v


absorcao_mais :: Def -> Def
absorcao_mais = penetra absor_mais
        where absor_mais ((g:\/:h):.:(i:+:j)) = ((g:.:i):\/:(h:.:j))
              absor_mais v = v

--catas
```

```
invabsorcao_mais :: Def -> Def
invabsorcao_mais z | penetra f z /= z = penetra invabsor_mais z
                   | otherwise = z
        where invabsor_mais (g:\/:(h:.:j)) = ((g:\/:h):.:(Id:+:j))
              invabsor_mais v = v
              f In = TRUE
              f z = z

invabsorcao_mais_invassociatividade d | x /= d && y/= x = (invabsorcao_mais.invassociatividade) d
                                      | otherwise = d
                      where x = invassociatividade d
                            y = invabsorcao_mais x


functor_mais :: Def -> Def
functor_mais = penetra funct_mais
        where funct_mais ((g:.:h):+:(i:.:j)) = ((g:+:i):.:(h:+:j))
              funct_mais v = v

functor_id_mais :: Def -> Def
functor_id_mais = penetra funct_id_mais
        where funct_id_mais (Id:+:Id) = Id
              funct_id_mais v = v
```

4 Outras Leis

```
------LEIS------


penetra :: (Exp -> Exp) -> Def -> Def
penetra p (f:&:g) = (penetra p f) :&: (penetra p g)
penetra p (f:=:g) = (penet p f) :=: (penet p g)
        where penet p e | p e /= e = p e
              penet p (f :/\: g) = (penet p f) :/\: (penet p g)
              penet p (f :\/: g) = (penet p f) :\/: (penet p g)
              penet p (f :*: g) = (penet p f) :*: (penet p g)
              penet p (f :+: g) = (penet p f) :+: (penet p g)
              penet p (f :.: g) = (penet p f) :.: (penet p g)
              penet p (f :@: g) = (penet p f) :@: (penet p g)
              penet p (Pair e1 e2) = Pair (penet p e1) (penet p e2)
              penet p (Cond f g h) = Cond (penet p f) (penet p g) (penet p h)
              penet p v = v


--
igualdade_either :: Def -> Def
igualdade_either ((f:=:g):&:(h:=:i)) = (f:\/:h) :=: (g:\/:i)
igualdade_either z = z

--meter ponto
composicao :: Def -> Def
composicao = penetra comp
```

```
        where comp (f:@:(g:@:x)) = (f:.:g):@:x
              comp v = v

--tirar ponto
invcomposicao :: Def -> Def
invcomposicao = penetra invcomp
        where invcomp ((f:.:g):@:x) = f:@:(g:@:x)
              invcomp v = v


associatividade :: Def -> Def
associatividade = penetra associa
        where associa ((f:.:g):.:h) = (f:.:(g:.:h))
              associa v = v

--catas
invassociatividade :: Def -> Def
invassociatividade = penetra invassocia
        where invassocia (f:.:(g:.:h)) = ((f:.:g):.:h)
              invassocia v = v

natural_id :: Def -> Def
natural_id = penetra nat_id
        where nat_id (f:.:Id) = f
              nat_id (Id:.:f) = f
              nat_id v = v


invnatural_id :: Def -> Def
invnatural_id = penetra invnat_id
        where invnat_id z@((f:.:g):/\:h) | g==Fst && h==Snd = ((f:.:g):/\:(Id:.:h))
                                            | otherwise = ((invnat_id f):.:(invnat_id g)):/\:(invnat_id h)
              invnat_id z@(h:/\:(f:.:g)) | g==Snd && h==Fst = ((Id:.:h):/\:(f:.:g))
                                         | otherwise = (invnat_id h):/\:((invnat_id f):.:(invnat_id g))
              invnat_id z@((f:.:g):\/:h) | f==Inl && h==Inr = ((f:.:g):\/:(h:.:Id))
                                            | otherwise = ((invnat_id f):.:(invnat_id g)):\/:(invnat_id h)
              invnat_id z@(h:\/:(f:.:g)) | f==Inr && h==Inl = ((h:.:Id):\/:(f:.:g))
                                         | otherwise = (invnat_id h):\/:((invnat_id f):.:(invnat_id g))
              invnat_id v = v



--tira a variavel
ext_igual :: Def -> Def
ext_igual (f:&:g) = (ext_igual f) :&: (ext_igual g)
ext_igual ((f:@:x):=:(g:@:y)) | x==y = f:=:g
ext_igual d = d


--poe a variavel
invext_igual :: Def -> Def
invext_igual (f:&:g) = (invext_igual f) :&: (invext_igual g)
invext_igual (f:=:g) | not (tem_var f) && not (tem_var g) && busca f == Id && busca g == Id = ((f :@: Var "x
invext_igual d = d

tem_var :: Exp -> Bool
tem_var a | elem "Var" ((lexL.show) a) = True
          | otherwise = False
```

```
--------Leis pwpf------------

substituir :: Def -> (Exp,Exp) -> Def
substituir (f:&:g) e = (substituir f e) :&: (substituir g e)
substituir (f:=:g) e = (subst f e) :=: (subst g e)
subst :: Exp -> (Exp,Exp) -> Exp
subst d (a,b) | d==a = b
subst (f :/\: g) e = (subst f e) :/\: (subst g e)
subst (f :\/: g) e = (subst f e) :\/: (subst g e)
subst (f :*: g) e = (subst f e) :*: (subst g e)
subst (f :+: g) e = (subst f e) :+: (subst g e)
subst (f :.: g) e = (subst f e) :.: (subst g e)
subst (f :@: g) e = (subst f e) :@: (subst g e)
subst (Pair e1 e2) e = Pair (subst e1 e) (subst e2 e)
subst (Cond f g h) e = Cond (subst f e) (subst g e) (subst h e)
subst (Const x) e = Const (subst x e)
subst v e = v


elim_x :: Def -> Def
elim_x (a:&:b) = (elim_x a):&:(elim_x b)
elim_x (a:=:b) = substituir subst1 y
        where (a1,(x,y)) = elimx2 a
              subst1 = substituir (a1:=:b) x


elimx2 :: Exp -> (Exp, ((Exp, Exp), (Exp, Exp)))
elimx2 p = runState (aux p) ((Id,Id),(Id,Id))
   where aux :: Exp -> State ((Exp,Exp),(Exp,Exp)) Exp
         aux (f :/\: g)   = do {a <- aux f; b <- aux g; return $ a :/\: b}
         aux (f :\/: g)   = do {a <- aux f; b <- aux g; return $ a :\/: b}
         aux (f :*: g)    = do {a <- aux f; b <- aux g; return $ a :*: b}
         aux (f :+: g)    = do {a <- aux f; b <- aux g; return $ a :+: b}
         aux (f :.: g)    = do {a <- aux f; b <- aux g; return $ a :.: b}
         aux (f :@: g)    = do {a <- aux f; b <- aux g; return $ a :@: b}
         aux (Cond f g h) = do {a <- aux f; b <- aux g; c <- aux h; return $ Cond a b c}
         aux (Const f)    = do {a <- aux f; return $ Const a}
         aux (Pair x y)   | ba x || ba y = do {a <- aux x; b <- aux y; return $ Pair a b}
                          | not (ba x) && not (ba y) = do {put ((x,Fst:@:xy),(y,Snd:@:xy)); return xy}
                                where  ba x = (head.lexL.show) x=="Pair"
                                       var = (read.last.lexL.show)
                                       xy = Var $ var x++var y
         aux z = return z



t2 = ((Fun "len" :@: (Const Nil :@: Var "x")) :=: (Const (Num 0) :@: Var "x")) :&: ((Fun "len" :@: (Const (

t = ((Fun "len" :@: (Const Nil :@: Var "x")) :=: (Const (Num 0) :@: Var "x")) :&: ((Fun "len" :@: (Fun "Cons


elim2 :: Exp -> (Exp, ((Exp, Exp), (Exp, Exp)))
```

```
elim2 p = runState (aux p) ((Id,Id),(Id,Id))
   where aux :: Exp -> State ((Exp,Exp),(Exp,Exp)) Exp
         aux e1 | ba e1 && (ba x || ba y) = do a <- aux x
                                              b <- aux y
                                              return $ Pair a b
                | ba e1 && not (ba x) && not (ba y) = do put ((x,Fst:@:xy),(y,Snd:@:xy))
                                                         return xy
                | otherwise = do return e1
                                 where  ba x = (head.lexL.show) x=="Pair"
                                        Pair x y = e1
                                        var = (read.last.lexL.show)
                                        xy = Var $ var x++var y

--Pair (Pair (Var "x") (Var "y")) (Var "z"))
-- ((Fun "ppu"):@: Var "k") :=: (Pair (Pair (Fst:@:(Fst:@:Var "k")) (Snd:@:(Snd:@:Var "k")) ) (Pair (Fst:@:

invelim_x :: Def -> Def
invelim_x z | not (elem "Var" ((lexL.show) z)) = z
            | elem ":&:" ((lexL.show) z) = (invelim_x q):&:(invelim_x w)
            | (not (elem "Var" ((lexL.show) e)) || not (elem "Var" ((lexL.show) e2))) = z
             | elem ":=:" ((lexL.show) z) && invelim2 e2 /=[] = f:@:(subst e1 (var1,(Pair var1 var2))) :=:(
            | otherwise = z
        where e:=:e2 = z
              f:@:e1 = e
              q:&:w = z
              var1 = head (invelim2 e2)
              var2 = Var (busca (da_var e1) (reverse ['p'..'z']))
              busca a [] = "?"
              busca a (x:xs) | elem [x] a = busca a xs
                             | otherwise = [x]


invelim2 :: Exp -> [Exp]
invelim2 e = (nub.aux.lexL.show) e
        where aux [] = []
              aux z | length z < 4 = []
              aux (x:y:z:w:xs) | (x=="Fst" || x=="Snd") && y==":@:" && z== "Var" = (Var (read w :: String))
                               | otherwise = aux (y:z:w:xs)


da_var :: Exp -> [String]
da_var e = nub.d_var $ ((lexL.show) e)
        where d_var [] = []
              d_var (x:xs) | "Var"==x = (read (head xs) :: String) : d_var xs
                           | otherwise = d_var xs




-- divisao lexica
lexL :: String -> [String]
lexL [] = []
lexL s = s1:(lexL s2)
    where [(s1,s2)] = lex s
```

5 Cata, Const, In, Bang, Nat-pi, Troca e Def-Id

```
def_inout :: Def -> Def
def_inout = penetra d_inout
        where d_inout (In:.:Out) = Id
              d_inout (Out:.:In) = Id
              d_inout z = z


meter_out :: Def -> Def
meter_out (f:&:g) = (meter_out f) :&: (meter_out g)
meter_out ((f:.:In):=:g) = ((f:.:In):.:Out):=:(g:.:Out)
meter_out ((In:.:f):=:g) = ((f:.:In):.:Out):=:(g:.:Out)
meter_out z = z



def_in :: Def -> Def
def_in = penetra d_in
        where d_in (Const Nil :\/: Cons) = In --RList
              d_in z = z

invdef_in :: Def -> Def
invdef_in = penetra invd_in
        where invd_in In = (Const Nil :\/: Cons) --RList
              invd_in z = z


def_cond :: Def -> Def
def_cond = penetra d_cond
        where d_cond (Cond f g h) = (g:\/:h):.:f
              d_cond z = z

def_cond2 :: Def -> Def
def_cond2 = penetra d_cond2
        where d_cond2 (Cond (f:@:x) (g:@:y) (h:@:z)) | x==y && y==z = (Cond f g h ):@: x
              d_cond2 z = z

invdef_cond2 :: Def -> Def
invdef_cond2 = penetra invd_cond2
        where invd_cond2 ((Cond f g h ):@: x) = (Cond (f:@:x) (g:@:x) (h:@:x))
              invd_cond2 z = z


--cata
def_cond3 :: Def -> Def
def_cond3 d | (penetra d_cond3 (invassociatividade d) /= invassociatividade d )
                                                                  = (penetra d_cond3).invassociatividade
```

```
        where d_cond3 (Cond (f:.:Fst) (g:.:(Id:*:y)) (h:.:(Id:*:z))) | y==z = (Cond (f:.:Fst) g h ):.: (Id
              d_cond3 (Cond (f:.:Snd) (g:.:(Id:*:y)) (h:.:(Id:*:z))) | y==z = (Cond (f:.:Snd) g h ):.: (Id
              d_cond3 z = z
def_cond3 d = d


--invcata
invdef_cond3 :: Def -> Def
invdef_cond3 d | (penetra invd_cond3 (associatividade d) /= associatividade d )
                                                                    = (penetra invd_cond3).associativida
        where invd_cond3 ((Cond (f:.:Fst) g h ):.: (Id :*: z)) = (Cond (f:.:Fst) (g:.:(Id:*:z)) (h:.:(Id:*:z
              invd_cond3 ((Cond (f:.:Snd) g h ):.: (Id :*: z)) = (Cond (f:.:Snd) (g:.:(Id:*:z)) (h:.:(Id:*:z
              invd_cond3 z = z
invdef_cond3 d = d




def_const :: Def -> Def
def_const = penetra d_const
        where d_const (Const x:@:y) = x
              d_const (Const x) = x
              d_const z = z




busca :: Exp -> Exp
busca p = snd (runState (aux p) Id)
   where aux :: Exp -> State Exp Exp
         aux (f :/\: g)   = do {a <- aux f; b <- aux g; return $ a :/\: b}
         aux (f :\/: g)   = do {a <- aux f; b <- aux g; return $ a :\/: b}
         aux (f :*: g)    = do {a <- aux f; b <- aux g; return $ a :*: b}
         aux (f :+: g)    = do {a <- aux f; b <- aux g; return $ a :+: b}
         aux (f :.: g)    = do {a <- aux f; b <- aux g; return $ a :.: b}
         aux (f :@: g)    = do {a <- aux f; b <- aux g; return $ a :@: b}
         aux (Cond f g h) = do {a <- aux f; b <- aux g; c <- aux h; return $ Cond a b c}
         aux (Const f)    = do {a <- aux f; return $ Const a}
         aux Nil              = do {put Nil; return Id}
         aux Unit             = do {put Unit; return Id}
         aux (Var x)             = do {put (Var x); return Id}
         aux (Num x)              = do {put (Num x); return Id}
         aux z = return z


invdef_const :: Def -> Def
invdef_const (f:&:g) = (invdef_const f) :&: (invdef_const g)
invdef_const (f:=:g) | v/= Id = (invd_const c f) :=: (invd_const c g)
                     | otherwise = (f:=:g)
                        where v = busca f
                              v2 = busca g
                              c = Var "c"

invd_const :: Exp -> Exp -> Exp
invd_const v Nil = Const Nil :@: v
invd_const v (Num x) = Const (Num x) :@: v
invd_const v Unit = v
```

```
invd_const v TRUE = Const TRUE :@: v
invd_const v FALSE = Const FALSE :@: v
invd_const v (f:\/:g) = (invd_const v f):\/:(invd_const v g)
invd_const v (f:/\:g) = (invd_const v f):/\:(invd_const v g)
invd_const v (f:*:g) = (invd_const v f):*:(invd_const v g)
invd_const v (f:+:g) = (invd_const v f):+:(invd_const v g)
invd_const v (f:.:g) = (invd_const v f):.:(invd_const v g)
invd_const v (Pair f g) = Pair (invd_const v f) (invd_const v g)
invd_const v (f:@:g) = (invd_const v f) :@: (invd_const v g)
invd_const v z = z
```

```
def_bang :: Def -> Def
def_bang = penetra d_bang
        where d_bang (Bang:@:x) = Unit
              d_bang v = v

invdef_bang :: Def -> Def
invdef_bang z | elem ":&:" ((lexL.show) z) = (invdef_bang q):&:(invdef_bang w)
              | elem ":=:" ((lexL.show) z) && (da_var e1/=[] || da_var e2/=[]) = (invd_bang (var e1) e1 ):=
              | otherwise = z
            where e1:=:e2 = z
                  q:&:w = z
                  var e = Var ((head.da_var) e)

invd_bang :: Exp -> Exp -> Exp
invd_bang v Unit = Bang :@: v
invd_bang v (f:\/:g) = (invd_bang v f):\/:(invd_bang v g)
invd_bang v (f:/\:g) = (invd_bang v f):/\:(invd_bang v g)
invd_bang v (f:*:g) = (invd_bang v f):*:(invd_bang v g)
invd_bang v (f:+:g) = (invd_bang v f):+:(invd_bang v g)
invd_bang v (f:.:g) = (invd_bang v f):.:(invd_bang v g)
invd_bang v (Pair f g) = Pair (invd_bang v f) (invd_bang v g)
invd_bang v (f:@:g) = (invd_bang v f) :@: (invd_bang v g)
invd_bang v z = z

--nat_fst, nat_snd
nat_pi :: Def -> Def
nat_pi = penetra n_pi
        where n_pi (Fst:.:(f:*:g)) = f:.:Fst
              n_pi (Snd:.:(f:*:g)) = g:.:Snd
              n_pi z = z


invnat_pi :: Def -> Def
invnat_pi (f:&:g) = (invnat_pi f) :&: (invnat_pi g)
invnat_pi ((f:.:In):=:g) = (f:.:In) :=: (invn_pi f g)
        where invn_pi p (f:.:Snd) | f==p = Snd:.:(Id:*:f)
              invn_pi p (f:.:Fst) | f==p = Fst:.:(f:*:Id)
              invn_pi p (f :/\: g) = (invn_pi p f) :/\: (invn_pi p g)
              invn_pi p (f :\/: g) = (invn_pi p f) :\/: (invn_pi p g)
              invn_pi p (f :*: g) = (invn_pi p f) :*: (invn_pi p g)
```

```
              invn_pi p (f :+: g) = (invn_pi p f) :+: (invn_pi p g)
              invn_pi p (f :.: g) = (invn_pi p f) :.: (invn_pi p g)
              invn_pi p (f :@: g) = (invn_pi p f) :@: (invn_pi p g)
              invn_pi p (Pair e1 e2) = Pair (invn_pi p e1) (invn_pi p e2)
              invn_pi p (Cond f g h) = Cond (invn_pi p f) (invn_pi p g) (invn_pi p h)
              invn_pi p v = v
invnat_pi z = z


def_fstsnd :: Def -> Def
def_fstsnd = penetra d_fs
        where d_fs (Fst :@:(Pair x y)) = x
              d_fs (Snd :@:(Pair x y)) = y
              d_fs v = v


def_fst :: Exp -> Exp
def_fst (Pair x y) = x

def_snd :: Exp -> Exp
def_snd (Pair x y) = y


def_id :: Def -> Def
def_id = penetra deff_id
        where deff_id (Id:@:f) = f
              deff_id v = v

troca :: Def -> Def
troca = penetra d_troca
        where d_troca ((f:/\:g):\/:(h:/\:i)) = ((f:\/:h):/\:(g:\/:i))
              d_troca ((f:\/:h):/\:(g:\/:i)) = ((f:/\:g):\/:(h:/\:i))
              d_troca v = v


troca1 :: Def -> Def
troca1 = penetra d_troca1
        where d_troca1 ((f:/\:g):\/:(h:/\:i)) = ((f:\/:h):/\:(g:\/:i))
              d_troca1 v = v

troca2 :: Def -> Def
troca2 = penetra d_troca2
        where d_troca2 ((f:\/:h):/\:(g:\/:i)) = ((f:/\:g):\/:(h:/\:i))
              d_troca2 v = v



-----CATAS-----

universal_cata :: Def -> Def
universal_cata z@((k:.:In):=:(b:.:(Id:+:(Id:*:f)))) | f==k = k:=: Cata b
universal_cata z = z

--pwpf def_len >>= return.universal_cata.invabsorcao_mais.invassociatividade.invnat_pi


invuniversal_cata :: Def -> Def
invuniversal_cata (k:=: Cata b) = ((k:.:In):=:(b:.:(Id:+:(Id:*:k))))
```

```
invuniversal_cata z = z

--pwpf def_soma >>= return.natural_id.absorcao_mais.invuniversal_cata
--pwpf def_len >>= return.invdef_in.nat_pi.associatividade.natural_id.absorcao_mais.invuniversal_cata
```

6 Funcoes de Conversao PWPF e PFPW

```
--------------------
--------------------
--PWPF--

leis_pwpf = ["troca1","troca2","invdef_x","invnatural_id","ext_igual","invdef_split","composicao","elim_x","

app_lei :: String -> Def -> Def
app_lei s d | s == "invdef_x" = invdef_x d
            | s == "invnatural_id" = invnatural_id d
            | s == "ext_igual" = ext_igual d
            | s == "invdef_split" = invdef_split d
            | s == "composicao" = composicao d
            | s == "elim_x" = elim_x d
            | s == "invuniversal_mais" = invuniversal_mais d
            | s == "invuniversal_x" = invuniversal_x d
            | s == "invdef_mais" = invdef_mais d
            | s == "invdef_bang" = invdef_bang d
--
            | s == "fusao_x" = fusao_x d
            | s == "invfusao_x" = fusao_x d
            | s == "reflexao_x" = reflexao_x d
            | s == "cancelamento_x" = cancelamento_x d
            | s == "fusao_mais" = fusao_mais d
            | s == "invfusao_mais" = invfusao_mais d
            | s == "reflexao_mais" = reflexao_mais d
            | s == "cancelamento_mais" = cancelamento_mais d
            | s == "functor_x" = functor_x d
            | s == "functor_id_x" = functor_id_x d
            | s == "functor_mais" = functor_mais d
            | s == "functor_id_mais" = functor_id_mais d
            | s == "absorcao_x" = absorcao_x d
            | s == "absorcao_mais" = absorcao_mais d
            | s == "associatividade" = associatividade d
--
            | s == "def_x" = def_x d
            | s == "natural_id" = natural_id d
            | s == "invext_igual" = invext_igual d
            | s == "def_split" = def_split d
            | s == "invcomposicao" = invcomposicao d
            | s == "invelim_x" = invelim_x d
            | s == "universal_mais" = universal_mais d
```

```
            | s == "universal_x" = universal_x d
            | s == "def_mais" = def_mais d
            | s == "def_bang" = def_bang d
--
            | s == "def_id" = def_id d
--            | s == "troca" = troca d  -- atencao: bipolar
            | s == "troca1" = troca1 d
            | s == "troca2" = troca2 d
            | s == "def_const" = def_const d
            | s == "invdef_const" = invdef_const d
--            | s == "def_cond" = def_cond d
            | s == "def_cond2" = def_cond2 d
            | s == "def_cond3" = def_cond3 d
            | s == "invdef_cond2" = invdef_cond2 d
            | s == "invdef_cond3" = invdef_cond3 d
            | s == "igualdade_either" = igualdade_either d
            | s == "def_inout" = def_inout d
            | s == "def_in" = def_in d
            | s == "invdef_in" = invdef_in d
            | s == "meter_out" = meter_out d
            | s == "universal_cata" = universal_cata d
            | s == "invuniversal_cata" = invuniversal_cata d
            | s == "invabsorcao_mais.invassociatividade" = (invabsorcao_mais_invassociatividade) d
            | s == "invabsorcao_mais" = invabsorcao_mais d
            | s == "invassociatividade" = invassociatividade d
            | s == "invnat_pi" = invnat_pi d
            | s == "nat_pi" = nat_pi d
            | s == "def_fstsnd" = def_fstsnd d

            | otherwise = d

pwpf :: Def -> IO Def
pwpf = pwpf' leis_pwpf

pwpf' :: [String] -> Def -> IO Def
pwpf' str d | d==z = return d
        where (z,s) = adiv_pwpf str (d,"")
pwpf' str d = do let (res,lei) = adiv_pwpf str (d,"")
                     putStrLn $ "{"++lei++"}"
                     putStrLn $ " => "++ show res++"\n"
                     let str2 = if(lei=="troca1" || lei=="troca2") then (((delete "troca2").(delete "troca1"
                     let str3 = if(lei=="fusao_mais" || lei=="invfusao_mais") then (((delete "fusao_mais").
                     let str4 = if(lei=="fusao_x" || lei=="invfusao_x") then (((delete "fusao_x").(delete "
                     let str5 = if(lei=="absorcao_mais" || lei=="invabsorcao_mais") then (((delete "absorca
                     let str6 = if(lei=="associatividade" || lei=="invassociatividade") then (((delete "asso
                 pwpf' str6 res



adiv_pwpf :: [String] -> (Def,String) -> (Def,String)
adiv_pwpf [] e = e
adiv_pwpf (x:xs) (e,s) | (app_lei x e) /= e && head lst=="Fun" = (app_lei x e,x)
                       | otherwise = adiv_pwpf xs (e,s)
                             where lst = ((filter (/="(")).lexL.show) (app_lei x e)



--PFPW--
```

```
leis_pfpw = ["invuniversal_cata","invdef_in","nat_pi","absorcao_mais","associatividade","universal_mais","u



{-
pfpw :: Def -> IO Def
pfpw d | d==z = return d
        where (z,s) = adiv_pwpf leis_pfpw (d,"")
pfpw d = do let (res,lei) = adiv_pwpf leis_pfpw (d,"")
            putStrLn $ "{"++lei++"}"
            putStrLn $ " => "++ show res++"\n"
            pfpw res
-}

pfpw :: Def -> IO Def
pfpw = pfpw' leis_pfpw

pfpw' :: [String] -> Def -> IO Def
pfpw' str d | d==z = return d
        where (z,s) = adiv_pwpf str (d,"")
pfpw' str d = do let (res,lei) = adiv_pwpf str (d,"")
                     putStrLn $ "{"++lei++"}"
                     putStrLn $ " => "++ show res++"\n"
                     let str2 = if(lei=="troca1" || lei=="troca2") then (((delete "troca2").(delete "troca1
                     let str3 = if(lei=="fusao_mais" || lei=="invfusao_mais") then (((delete "fusao_mais").
                     let str4 = if(lei=="fusao_x" || lei=="invfusao_x") then (((delete "fusao_x").(delete "
                 pfpw' str4 res
```