

Cálculo Lambda em Haskell

Rodrigo Machado

23 de agosto de 2012

Este documento reporta os passos necessários para a construção de um interpretador com interface gráfica para cálculo lambda usando a linguagem Haskell.

1 Sintaxe

Definição do módulo principal e bibliotecas utilizadas.

```
module Main where
import Data.List (union,\\)
import Data.IORef
import qualified Data.ByteString as BS
import qualified Data.Text as T
import qualified Data.Text.Encoding as TE
import qualified Data.Text.Encoding.Error as TEE
import qualified Control.Exception as E
import Control.Monad (replicateM_)
import Graphics.UI.Gtk
import Graphics.UI.Gtk.Builder
import Graphics.Rendering.Cairo
import Graphics.Rendering.Pango.Layout
import Text.ParserCombinators.Parsec
```

O primeiro passo é descrever um tipo de dados para representar termos lambda (sintaxe abstrata). Para variáveis, nós utilizaremos `Id` como um sinônimo para o tipo pré-definido `String`.

```
type Id = String
data Term = Var Id
          | Lambda Id Term
          | App Term Term
          deriving (Eq,Ord,Show,Read)
```

2 Ocorrência de variáveis

Esta seção define funções que lidam com os conceitos de variável livre e ligada, como segue:

- `vars`: extrai todas as variáveis que ocorrem no termo.

```
vars :: Term -> [Id]
vars (Var x)      = [x]
vars (Lambda x t) = [x] 'union' (vars t)
vars (App t1 t2)  = (vars t1) 'union' (vars t2)
```

- `fv`: extrai todas as variáveis que ocorrem livres no termo.

```
fv :: Term -> [Id]
fv (Var x)      = [x]
fv (Lambda x t) = (fv t) \ [x]
fv (App t1 t2)  = (fv t1) 'union' (fv t2)
```

3 Substituição

Agora definimos substituição de variáveis. A chamada $(\text{sub } x \ v \ t)$ pode ser lido como $[x := v](t)$, isto é, substituir em t todas as ocorrências livres da variável x pelo termo v .

```
sub :: Id -> Term -> Term -> Term
sub x v (Var y)      | y == x    = v
                    | otherwise = Var y
sub x v (Lambda y t) | y == x    = Lambda y t
                    | otherwise = Lambda y (sub x v t)
sub x v (App t1 t2)  = App (sub x v t1) (sub x v t2)
```

A função `sub`, contudo, não evita a possível captura de variáveis livres em v . Por exemplo:

$$\text{sub } y \ x \ (\lambda x.x \ y) = \lambda x.x \ x$$

Na posição da variável livre y (em $\lambda x.x \ y$), o nome x está ligado. Ao realizar a substituição, a variável livre x (segundo parâmetro) acaba se ligando, isto é se confundindo com o parâmetro formal x (em $\lambda x.x \ y$). Para evitar essa situação, é necessário poder realizar a troca do nome de variáveis ligadas.

4 Redução alfa

A troca de nomes de variáveis ligadas é obtida em cálculo lambda através da operação de redução alfa. Na implementação em Haskell, a função `alpha` recebe uma lista de variáveis que não podem ser usadas como variáveis ligadas e um termo.

```
alpha :: [Id] -> Term -> Term
alpha xs (Var y)      = Var y
alpha xs (Lambda y t) | y 'elem' xs = let n = newName xs in
                                      Lambda n (sub y (Var n) (alpha (n:xs) t))
                    | otherwise    = Lambda y (alpha (y:xs) t)
alpha xs (App t1 t2)  = App (alpha xs t1) (alpha xs t2)
```

A lista infinita `names` contém strings que podem ser usadas como variáveis. A função `newName`, chamada por `alpha`, cria novos nomes para ligações no termo, certificando-se que eles não ocorrem na lista de nomes recebida. Para tal, ela retorna o primeiro elemento de nomes que não esteja na lista recebida.

```
names :: [Id]
names = tail $ gen [[]]
      where gen x = x ++ gen [ c:s | c <- ['a'..'z'], s <- x ]

newName :: [Id] -> Id
newName xs = head $ filter ('notElem' xs) names
```

Se nos certificarmos que não haverá nenhuma variável ligada com mesmo nome que uma variável livre, a operação de substituição se torna segura. Por exemplo:

$$\begin{aligned}\text{alpha } [x] (\lambda x. x y) &= (\lambda a. a y) \\ \text{sub } y x (\lambda a. a y) &= (\lambda a. a x)\end{aligned}$$

5 Formas normais

Um *redex* (“reducible expression”) é um termo lambda na forma $(\lambda x. M)N$. Um termo lambda é uma *forma normal* se não for um redex e não possuir nenhum redex como subtermo. A chamada $(\text{nf } t)$ testa se t é uma forma normal.

```
nf :: Term -> Bool
nf (Var _)      = True
nf (Lambda x t) = nf t
nf (App (Lambda _ _) _) = False
nf (App t1 t2)  = (nf t1) && (nf t2)
```

6 Redução beta

Utilizando as funções alpha e sub , podemos finalmente definir a função de redução beta. O tipo de retorno de beta é Maybe Term pois é possível que o termo a ser avaliado seja uma forma normal (irredutível). Note que beta utiliza uma estratégia preguiçosa de avaliação, isto é, redução sempre pelo redex mais externo.

```
beta :: Term -> Maybe Term
beta (Var x)      = Nothing
beta (Lambda x t) = do {t' <- beta t; return (Lambda x t')}
beta t@(App (Lambda x t1) t2) = Just $ sub x t2 (alpha (fv t) t1)
beta (App t1 t2) | not (nf t1) = do {t1' <- beta t1; return (App t1' t2)}
                  | otherwise  = do {t2' <- beta t2; return (App t1 t2')}
```

Podemos também definir uma versão estrita de avaliação. Função betaStrict avalia a função e os argumentos até a forma normal (se houver) antes de reduzir o redex.

```
betaStrict :: Term -> Maybe Term
betaStrict (Var x)      = Nothing
betaStrict (Lambda x t) = do {t' <- betaStrict t; return (Lambda x t')}
betaStrict t@(App t1 t2) | not (nf t1) = do {t1' <- betaStrict t1; return (App t1' t2)}
                          | not (nf t2) = do {t2' <- betaStrict t2; return (App t1 t2')}
                          | otherwise  = case t1 of
                                          Lambda x t3 -> Just $ sub x t2
                                                                (alpha (fv t) t3)
                                          otherwise  -> Nothing
```

7 Impressão legível e parser

Esta seção define funções de tradução entre a sintaxe concreta de termos lambda e a sintaxe abstrata. A primeira função (pretty) permite a visualização mais intuitiva de termos, convertendo-os da sintaxe abstrata para a sintaxe concreta.

```
pretty :: Term -> String
```

pretty (Var s)	= s
pretty (Lambda s t)	= "\\\" ++ s ++ \".\" ++ pretty t
pretty (App t1@(App _ _) t2@(App _ _))	= pretty t1 ++ \" (\" ++ pretty t2 ++ \")\"
pretty (App t1@(App _ _) t2@(Lambda _ _))	= pretty t1 ++ \" (\" ++ pretty t2 ++ \")\"
pretty (App t1@(App _ _) t2)	= pretty t1 ++ \" \" ++ pretty t2
pretty (App t1@(Var _) t2@(Var _))	= pretty t1 ++ \" \" ++ pretty t2
pretty (App t1@(Var _) t2)	= pretty t1 ++ \" (\" ++ pretty t2 ++ \")\"
pretty (App t1 t2@(Var _))	= \"(\" ++ pretty t1 ++ \") \" ++ pretty t2
pretty (App t1 t2)	= \"(\" ++ pretty t1 ++ \") (\" ++ pretty t2 ++ \")\"

A outra função (pparser) é um parser de termos lambda construído através da biblioteca Parsec. A sintaxe concreta está disponível visualmente na interface gráfica. Note que o parser lê os termos lambda seguindo as convenções sintáticas habituais: aplicação é associativa à esquerda, maior escopo possível para abstração lambda.

```
termP :: Parser Term
termP = do spaces
        (h:t) <- sepEndBy1 (lambP <|> varP <|> numP <|> parenP ) spaces
        spaces
        return (foldl App h t) -- aplicacao

lambP = do char '\\\'
        spaces
        xs <- sepEndBy1 idP spaces
        spaces
        char \'.\'
        spaces
        t <- termP
        spaces
        return (chain xs t) -- lambda
    where
        chain (a:b) t = Lambda a (chain b t)
        chain []      t = t

varP = do x <- idP
        spaces
        return (Var x) -- variaveis

numP = do xs <- many1 digit
        spaces
        return (cn (read xs))

parenP = do char \'(\'
        t <- termP
        char \')\'
        spaces
        return t -- parenteses

idP :: Parser String
idP = do x <- (letter <|> char \'_\')
        y <- many (letter <|> digit <|> char \'_\')
        return (x:y) -- identificadores
```

Por conveniência, o parser termP aceita que o usuário escreva número naturais na posição de um termo. Esses números são codificados como termos lambda pela função cn definida a seguir.

```
-- Funcao que constroi o numeral de Church a partir de um inteiro
cn :: Integer -> Term
cn n = Lambda "f" (Lambda "x" (rec n))
```

```

where rec n | n <= 0    = (Var "x")
      | otherwise = (App (Var "f") (rec (n-1)))

```

Outra conveniência é a apresentação de termos lambda como numerais de Church. Para tal, a função numRep “decodifica” um numeral de Church, e o apresenta como um número natural. Caso o termo seja mal-formado, então Nothing é retornado.

```

-- Funcao que retorna o número representado pelo termo, ou Nothing
-- se o termo for mal-formado
numRep :: Term -> Maybe Integer
numRep (Var _)    = Nothing
numRep (App _ _) = Nothing
numRep (Lambda a (Lambda b t)) = collect 0 a b t
  where
    collect x i j (App (Var k) u) | i == k    = collect (x+1) i j u
                                   | otherwise = Nothing
    collect x i j (Var k)         | j == k    = Just x
                                   | otherwise = Nothing
    collect x i j _              = Nothing
numRep (Lambda _ _) = Nothing

```

Comentários são iniciados pelo código -- e são finalizados ao término da linha.

```

commentP = do string "--"
            manyTill anyChar (try (string "\n"))

```

Programas são sequências de definições finalizadas por um termo. O parser progP lê programas inteiros, construindo um termo lambda único ao final do processo.

```

progP :: Parser Term
progP = do spaces
        (<|>) (do {x<-commentP;y<-progP;return y})
        ((<|>) (do {s <- idP; spaces;
                    char '='; spaces;
                    t <- termP; spaces;
                    char ',';
                    u <- progP;
                    return (App (Lambda s u) t)}}
            (do {string ">>"; u<-termP; return u})))

-- conta número de definições no texto
defnP :: Parser Integer
defnP = do spaces
        (<|>) (do {x<-commentP;y<-defnP;return y})
        ((<|>) (do {s <- idP; spaces;
                    char '='; spaces;
                    t <- termP; spaces;
                    char ',';
                    u <- defnP;
                    return (u+1)}}
            (do {string ">>"; u<-termP; return 0})))

```

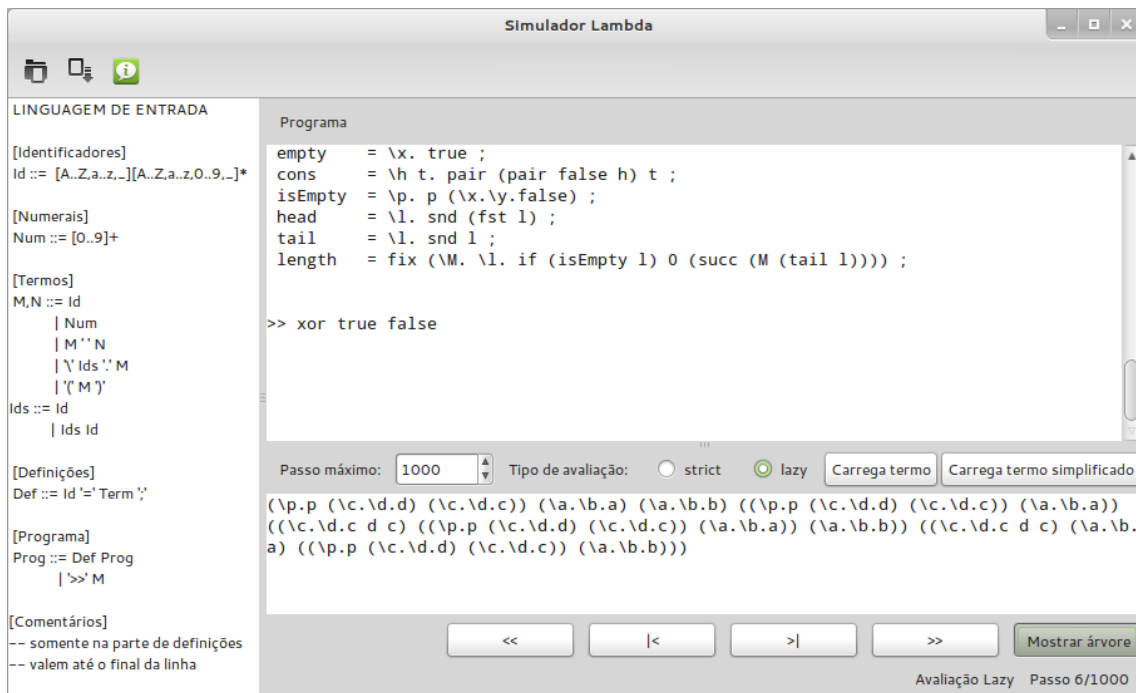


Figura 1: Interface principal.

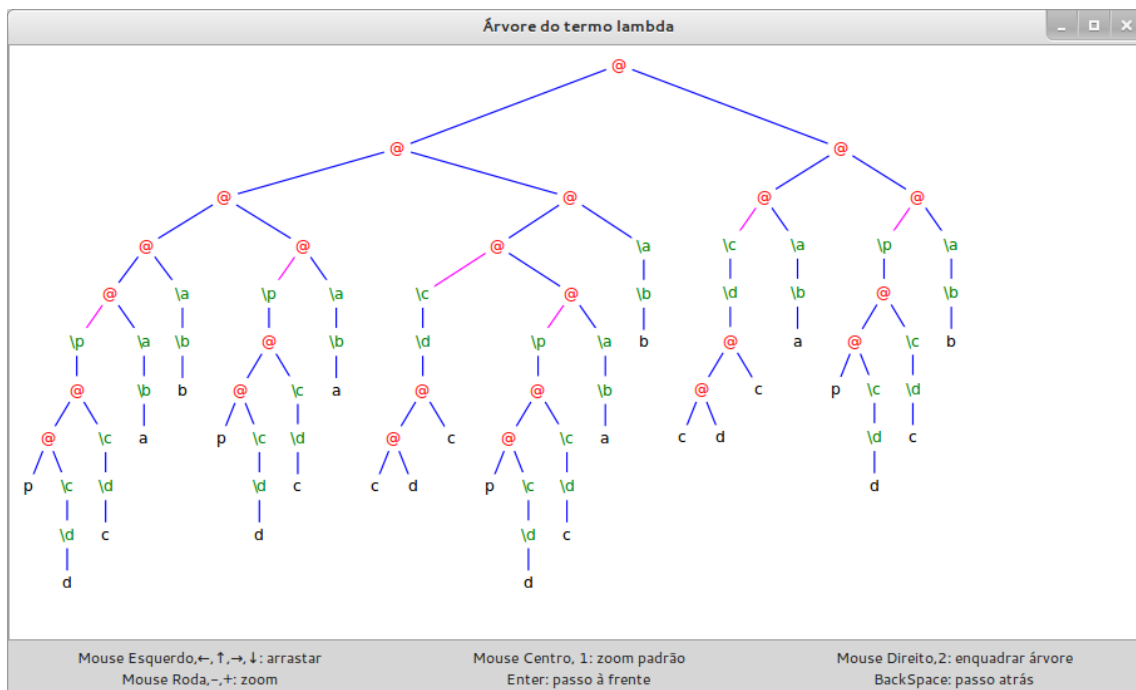


Figura 2: Janela de visualização da árvore de sintaxe.

8 Função principal e interface

A função principal carrega a interface gráfica mostrada pelas Figuras 1 e 2. A interface foi desenvolvida com a ferramenta Glade e é carregada dinamicamente a cada execução. Isto requer que o arquivo `lambda.ui` esteja no mesmo diretório do executável no momento da invocação.

```
main = do
  -- Inicializa Gtk
  initGUI

  -- Carrega arquivo de interface
  b <- builderNew
  builderAddFromFile b "./lambda.ui"

  ----- Preparação da Janela 1 -----
  window <- builderGetObject b castToWindow      "window1"
  text1  <- builderGetObject b castToTextView   "textview1"
  text2  <- builderGetObject b castToTextView   "textview2"
  button1 <- builderGetObject b castToButton     "button1"
  button2 <- builderGetObject b castToButton     "button2"
  button3 <- builderGetObject b castToButton     "button3"
  button4 <- builderGetObject b castToButton     "button4"
  button5 <- builderGetObject b castToButton     "button5"
  button6 <- builderGetObject b castToButton     "button6"
  tbutton <- builderGetObject b castToToggleButton "togglebutton1"
  spin1  <- builderGetObject b castToSpinButton  "spinbutton1"
  label3 <- builderGetObject b castToLabel       "label3"
  radio1 <- builderGetObject b castToRadioButton "radiobutton1"
  radio2 <- builderGetObject b castToRadioButton "radiobutton2"
  buf1   <- get text1 textViewBuffer
  buf2   <- get text2 textViewBuffer
  tool2  <- builderGetObject b castToToolButton  "toolbutton2"
  tool3  <- builderGetObject b castToToolButton  "toolbutton3"
  tool4  <- builderGetObject b castToToolButton  "toolbutton4"
  aboutD <- builderGetObject b castToAboutDialog "aboutdialog1"

  -- Define fontes das caixas de texto
  srcfont <- fontDescriptionFromString "monospace 10"
  widgetModifyFont text1 (Just srcfont)
  widgetModifyFont text2 (Just srcfont)

  -- Define estado da simulação:

  -- Estado = (termo atual (maybe), termos anteriores, passo atual, passo maximo, avaliacaoLazy?)
  state <- newIORef $ (Nothing, [], 0, 0, True)

  ----- Preparação da Janela 2 -----
  window2 <- builderGetObject b castToWindow      "window2"
  da      <- builderGetObject b castToDrawingArea "drawingarea1"

  -- Define fontes da janela de visualização de árvore
  dafont <- fontDescriptionFromString "sans 12"
  widgetModifyFont da (Just dafont)

  -- Prepara área de desenho
  widgetModifyBg da StateNormal (Color 65535 65535 65535)
  widgetAddEvents da [AllEventsMask]
```

```

-- Define estado da visualização:

-- Escala e translação da árvore de desenho
scale      <- newIORef 1.0
translation <- newIORef (0,0)

-- Ponto para controle de translação (maybe)
oldPoint   <- newIORef $ Just (0,0)

-- Conexão para atualizar desenho (inicialmente vazia)
c <- da 'on' exposeEvent $ return True
conn <- newIORef c

----- Registro de tratadores de evento (Janela 2) -----

-- Processa evento de rolagem da roda do mouse
da 'on' scrollEvent $ do
  dir <- eventScrollDirection
  liftIO $ do
    case dir of
      ScrollDown -> liftIO $ modifyIORef scale (*0.9)
      ScrollUp   -> liftIO $ modifyIORef scale (*1.1)
    widgetQueueDraw da
  return True

-- Processa dados do teclado
da 'on' keyPressEvent $ do
  k <- eventKeyName
  liftIO $ do
    z <- readIORef scale
    case k of
      "plus" -> do modifyIORef scale (*1.1)
                  widgetQueueDraw da
      "minus" -> do modifyIORef scale (*0.9)
                  widgetQueueDraw da
      "1"      -> do writeIORef scale 1
                  widgetQueueDraw da
      "2"      -> adjustZoom da state scale translation oldPoint
      "Up"      -> do modifyIORef translation \(a,b)-> (a,b+5/z))
                  widgetQueueDraw da
      "Down"    -> do modifyIORef translation \(a,b)-> (a,b-5/z))
                  widgetQueueDraw da
      "Left"    -> do modifyIORef translation \(a,b)-> (a+5/z,b))
                  widgetQueueDraw da
      "Right"   -> do modifyIORef translation \(a,b)-> (a-5/z,b))
                  widgetQueueDraw da
      "KP_Enter" -> stepForward buf2 label3 state tbutton da conn scale translation
      "Return"   -> stepForward buf2 label3 state tbutton da conn scale translation
      "BackSpace" -> stepBack buf2 label3 state tbutton da conn scale translation
      _          -> return ()
  return True

-- Processa botão do mouse sendo pressionado
da 'on' buttonPressEvent $ do
  (x,y) <- eventCoordinates
  b     <- eventButton
  liftIO $
    case b of

```



```

-- registra o ponto onde o botão foi pressionado
LeftButton -> writeIORef oldPoint $ Just (x,y)

- - - - -
-> return ()
return True

-- Processa botão do mouse sendo liberado
da 'on' buttonReleaseEvent $ do
  b <- eventButton
  liftIO $
    case b of

      -- apaga o ponto armazenado anteriormente
      LeftButton -> writeIORef oldPoint $ Nothing

      -- aplica zoom 1:1
      MiddleButton -> do writeIORef scale 1
                        widgetQueueDraw da

      -- enquadra árvore na janela
      RightButton -> adjustZoom da state scale translation oldPoint

    - - - - -
    -> return ()
  return True

-- Processa movimentação do mouse
da 'on' motionNotifyEvent $ do
  (x,y) <- eventCoordinates
  m <- eventModifierAll
  liftIO $
    if Button1 'elem' m
    then do
      -- lê ponto onde mouse foi pressionado e escala
      o <- readIORef oldPoint
      z <- readIORef scale
      case o of
        Just (x',y') -> do

          -- atualiza translação da janela
          modifyIORef translation (\(a,b)->(a+(x-x')/z,b+(y-y')/z))

          -- atualiza ponto anterior para o atual
          writeIORef oldPoint $ Just (x,y)

          widgetQueueDraw da

        Nothing -> return ()
      else return ()
    return True

----- Registro de tratadores de evento (Janela 1) -----

-- Alterna forma de avaliação (lazy vs strict)
radio1 'on' toggled $ switchStrategy radio1 state
radio2 'on' toggled $ switchStrategy radio1 state

-- Mostra/esconde janela 2
tbutton 'on' toggled $ do
  s <- toggleButtonGetActive tbutton
  case s of
    True -> do showTree scale translation conn window2 da state oldPoint

```

```

        widgetShowAll window2
        False -> widgetHideAll window2

-- Carrega programa processando definições
button1 'on' buttonActivated $
    loadProgramSimple buf1 buf2 label3 radio1 spin1 state tbutton da conn scale translation

-- Carrega programa sem processar definições
button4 'on' buttonActivated $
    loadProgram buf1 buf2 label3 radio1 spin1 state tbutton da conn scale translation

-- Um passo para trás na avaliação (quando possível)
button2 'on' buttonActivated $
    stepBack buf2 label3 state tbutton da conn scale translation

-- Um passo para frente na avaliação (quando possível)
button3 'on' buttonActivated $
    stepForward buf2 label3 state tbutton da conn scale translation

-- Volta para o primeiro termo da avaliação
button5 'on' buttonActivated $
    runBack buf2 label3 state tbutton da conn scale translation

-- Corre avaliação até limite máximo de passos ou até chegar a uma forma normal
button6 'on' buttonActivated $
    runForward buf2 label3 state tbutton da conn scale translation

----- Registro de tratadores de evento (Janela 1, barra de tarefas) -----

-- Botão "Sobre"
onToolButtonClicked tool4 $ do dialogRun aboutD
                             widgetHide aboutD

-- Botão "Abrir arquivo"
onToolButtonClicked tool2 $ abreArquivo window buf1

-- Diálogo "Salvar arquivo"
onToolButtonClicked tool3 $ salvaArquivo window buf1

-- Inicialização e destruição de janelas
onDestroy window $ do widgetDestroy window2
                    widgetDestroy aboutD
                    mainQuit
onDelete window2 $ (\e -> do widgetHide window2
                        toggleButtonSetActive tbutton False
                        return True)
onDelete aboutD $ (\e -> do widgetHide aboutD
                        return True)

-- Condição inicial das janelas
widgetShowAll window
widgetHideAll window2

-- Executa loop principal
mainGUI

----- Helpers -----

-- Mostra mensagem de alerta
alert :: String -> IO ()

```

```

alert s = do dia <- messageDialogNew Nothing [DialogModal] MessageInfo ButtonsOk s
            dialogRun dia
            widgetDestroy dia
            return ()

----- Tratadores de eventos -----

-- Abre arquivo
abreArquivo window buf1 = do
    openD <- fileChooserDialogNew
        (Just "Abrir arquivo")
        (Just window)
        FileChooserActionOpen
        [("Cancela",ResponseCancel),("Abre",ResponseAccept)]
    fileChooserSetDoOverwriteConfirmation openD True
    widgetShow openD
    response <- dialogRun openD
    case response of
        ResponseAccept -> do
            filename <- fileChooserGetFilename openD
            case filename of
                Nothing -> widgetHide openD
                Just path -> do
                    tentativa <- E.try (BS.readFile $ path) :: IO (Either E.IOException BS.ByteString)
                    case tentativa of
                        Left _ -> alert "Não foi possível ler arquivo"
                        Right conteudo -> do
                            let conteudo' = T.unpack $ TE.decodeUtf8With TEE.lenientDecode conteudo
                            set buf1 [textBufferText := conteudo']
            _ -> widgetHide openD
    widgetDestroy openD

-- Salva arquivo
salvaArquivo window buf1 = do
    saved <- fileChooserDialogNew
        (Just "Salvar arquivo")
        (Just window)
        FileChooserActionSave
        [("Cancela",ResponseCancel),("Salva",ResponseAccept)]
    fileChooserSetDoOverwriteConfirmation saved True
    widgetShow saved
    response <- dialogRun saved
    case response of
        ResponseAccept -> do
            filename <- fileChooserGetFilename saved
            case filename of
                Nothing -> widgetHide saved
                Just path -> do
                    conteudo <- get buf1 textBufferText
                    tentativa <- E.try (writeFile path conteudo) :: IO (Either E.IOException ())
                    case tentativa of
                        Left _ -> alert "Não foi possível escrever no arquivo"
                        Right _ -> return ()
            _ -> widgetHide saved
    widgetDestroy saved

-- Carrega programa e o transforma em termo lambda (sem rodar as definições)

```

```

loadProgram buf1 buf2 label3 radiol spin1 state tbutton da conn scale translation = do
  s1 <- get buf1  textBufferText
  ev <- get radiol toggleButtonActive
  mv <- get spin1 spinButtonValue
  case (parse progP "" s1) of
    Left _ -> writeIORef state (Nothing, [], 0, 0, True)
    Right t -> writeIORef state (Just t, [], 0, floor mv, ev)
  updateGUI buf2 label3 state tbutton da conn scale translation

-- Carrega programa e o transforma em termo lambda (substituindo definições)

loadProgramSimple buf1 buf2 label3 radiol spin1 state tbutton da conn scale translation = do
  s1 <- get buf1  textBufferText
  ev <- get radiol toggleButtonActive
  mv <- get spin1 spinButtonValue
  case (parse progP "" s1, parse defnP "" s1) of
    (Left _,_) -> writeIORef state (Nothing, [], 0, 0, True)
    (_,Left _) -> writeIORef state (Nothing, [], 0, 0, True)
    (Right t,Right n) -> do writeIORef state (Just t, [], 0, floor mv, True) -- set Lazy
                          replicateM_ (fromIntegral n)
                                (stepForward buf2 label3 state tbutton da conn scale translation)
                          modifyIORef state (\(a,_,c,m,_)>(a,[],0,floor mv,ev))
  updateGUI buf2 label3 state tbutton da conn scale translation

-- Passo para trás na execução

stepBack buf2 label3 state tbutton da conn scale translation = do
  (a,b,c,m,e) <- readIORef state
  case (a,b) of
    (Just t, h:d) -> writeIORef state (Just h,d,c-1,m,e)
    _ -> return ()
  updateGUI buf2 label3 state tbutton da conn scale translation

-- Passo para frente na execução

stepForward buf2 label3 state tbutton da conn scale translation = do
  (a,b,c,m,e) <- readIORef state
  let ev = if e then beta else betaStrict
  case do { t<-a; h<-ev t; return (t,h, c < m) } of
    Just(t,h,True) -> writeIORef state (Just h,t:b,c+1,m,e)
    _ -> return ()
  updateGUI buf2 label3 state tbutton da conn scale translation

-- Retorna ao início da execução

runBack buf2 label3 state tbutton da conn scale translation = do
  (a,b,c,m,e) <- readIORef state
  case (a,b) of
    (_,[]) -> return ()
    (Nothing,_) -> return ()
    (Just t,l) -> do let x = last l
                    modifyIORef state (\(a,b,c,m,e) -> (Just x, [], 0, m, e))
                    updateGUI buf2 label3 state tbutton da conn scale translation

-- Executa até chegar a forma normal ou ao limite de passos

```

```

runForward buf2 label3 state tbutton da conn scale translation = do
  (a,b,c,m,e) <- readIORef state
  case do {t<-a; return (t, not (nf t), c < m)} of
    Just (t,True,True) -> do stepForward buf2 label3 state tbutton da conn scale translation
                          runForward buf2 label3 state tbutton da conn scale translation
    _                    -> updateGUI buf2 label3 state tbutton da conn scale translation

-- Exibe o estado atual nas janelas 1 e 2

updateGUI buf2 label3 state tbutton da conn scale translation = do
  (a,b,c,m,e) <- readIORef state
  case a of
    Just t -> do set buf2 [textBufferText := pretty t]
                  set label3 [labelText := l1 ++ l2 ++ "Passo " ++
                                (show c) ++ "/" ++ show m]
                  -- testa se deve atualizar árvore
                  status <- toggleButtonGetActive tbutton
                  if status
                    then do -- disconnect old handler connection
                          s <- readIORef conn
                          signalDisconnect s
                          -- process exposure event
                          s' <- da 'on' exposeEvent $ do
                                d <- eventWindow
                                liftIO $ do
                                  drawWindowClear d
                                  zoom <- readIORef scale
                                  trans <- readIORef translation
                                  tbb <- toTermBBox t da
                                  renderWithDrawable d (renderTermB tbb zoom trans da)
                                return True
                          -- store new handler connection
                          writeIORef conn s'
                          -- ask for redraw of drawing area
                          widgetQueueDraw da
                    else return ()
    where l1 = case numRep t of
                  Nothing -> ""
                  Just x -> "(CN " ++ (show x) ++ " )      "
          l2 = if e then "Avaliação Lazy      "
                else "Avaliação Strict      "
    Nothing -> do set buf2 [textBufferText := "Programa mal-formado"]
                  set label3 [labelText := ""]

-- Alterna entre as estratégias lazy e strict

switchStrategy r1 state = do
  (a,b,c,m,e) <- readIORef state
  isLazy <- get r1 toggleButtonActive
  writeIORef state (a,b,c,m,isLazy)

-- Atualiza visualização da janela 2

showTree scale translation conn window2 da state oldPoint = do
  (a,b,c,m,e) <- readIORef state
  case a of

```

```

Nothing -> return ()
Just t -> do

    -- disconnect old handler connection
    s <- readIORef conn
    signalDisconnect s

    -- process exposure event
    s' <- da 'on' exposeEvent $ do
        d <- eventWindow
        liftIO $ do
            drawWindowClear d
            zoom <- readIORef scale
            trans <- readIORef translation
            tbb <- toTermBBox t da
            renderWithDrawable d (renderTermB tbb zoom trans da)

        return True

    -- store new handler connection
    writeIORef conn s'

    -- set visibility ON
    widgetShowAll window2

    -- ask for redraw of drawing area
    widgetQueueDraw da

    -- adjust zoom to make tree fit into window
    adjustZoom da state scale translation oldPoint

    return ()

-- Ajusta zoom
adjustZoom da state scale translation oldPoint = do
    (w,h) <- widgetGetSize da
    z <- readIORef scale
    (a,_,_,_,_) <- readIORef state
    case a of
        Just t -> do tbb <- toTermBBox t da
            let (BBox _ w' h' _) = getBBox $ tbb
                newZoom = min (min (fromIntegral w/w') (fromIntegral h/h')) 2
            -- reset translation
            writeIORef translation (0,0)
            writeIORef oldPoint $ Just (0,0)
            -- apply zoom
            writeIORef scale newZoom
            widgetQueueDraw da
        Nothing -> return ()

```

9 Representação gráfica dos termos

As seguintes funções implementam a conversão de um termo lambda em uma árvore visual. Como as árvores de termos lambda podem ser muito complexas, a ideia é calcular o espaço necessário para cada subárvore de cima para baixo, isto é, começar nas folhas (variáveis) e ir alocando espaço em direção à raiz da árvore. Para tal, fazemos

uso de um estrutura de dados que representa *bounding boxes*, isto é, caixas de contorno.

```
-- Estrutura de dados para Bounding Boxes
data BBox = BBox Double Double Double Double deriving (Eq,Ord,Read,Show)

-- BBox r w h h':
-- r = fração entre 0 e 1 representando onde está a raiz da árvore horizontalmente
-- w = largura da caixa
-- h = altura da caixa
-- h' = altura da parte da raiz

-- Acesso aos componentes da caixa
bbratio    (BBox r _ _ _) = r
bbwidth    (BBox _ w _ _) = w
bbheight   (BBox _ _ h _) = h
bbrootheight (BBox _ _ _ h') = h'
```

A seguinte estrutura de dados consiste de uma árvore de termos anotada com a respectiva caixa de contorno. A ideia é que este termo anotado seja calculado uma vez a partir de um termo comum, e seja usado para renderizar árvores de forma eficiente.

```
data TermBBox = VarB Id BBox
              | LambdaB Id TermBBox BBox
              | AppB TermBBox TermBBox BBox
              deriving (Eq,Ord,Show,Read)

getBBox :: TermBBox -> BBox
getBBox (VarB _ b)      = b
getBBox (LambdaB _ _ b) = b
getBBox (AppB _ _ b)    = b
```

Precisamos definir alguns parâmetros globais para que determinam o tamanho das caixas para as folhas da árvore (variáveis).

```
-- Altura mínima entre nodos da árvore
defh = 50
```

A Função a seguir converte um termo em um termo anotado, a ser utilizado na renderização.

```
-- função auxiliar para expandir bboxes de filhos.
-- Necessário quando, em um termo (Lambda s t) temos a renderização de "\s" mais largo que t
pad n (VarB s (BBox r w h h')) = (VarB s (BBox r (w+n) h h'))
pad n (LambdaB s t (BBox r w h h')) = (LambdaB s (pad n t) (BBox r (w+n) h h'))
pad n (AppB t t' (BBox r w h h')) = (AppB (pad (n/2) t) (pad (n/2) t') (BBox r (w+n) h h'))

-- toTermBBox converte um termo lambda em um termo lambda anotado com caixas
-- utiliza Pango para calcular tamanho das caixas de texto.

toTermBBox :: Term -> DrawingArea -> IO TermBBox

toTermBBox (Var s) da = do
  ctx <- widgetGetPangoContext da
  pL <- layoutText ctx s
  (_,PangoRectangle px py pw ph) <- layoutGetExtents pL
  return $ VarB s (BBox 0.5 (max (pw+10) 40) (ph+10+defh) (defh))
```

```

toTermBBox (Lambda s x) da = do
  subterm <- toTermBBox x da
  let BBox r w h h' = getBBox subterm
  ctx <- widgetGetPangoContext da
  pL <- layoutText ctx $ "\"" ++ s
  (_,PangoRectangle px py pw ph) <- layoutGetExtents pL
  if (pw+10) > w
    then return $ LambdaB s (pad ((pw+10)-w) subterm) (BBox r (pw+10) (h + defh) defh)
    else return $ LambdaB s subterm (BBox r w (h + defh) defh)

toTermBBox (App x y) da = do
  subterm1 <- toTermBBox x da
  subterm2 <- toTermBBox y da
  let BBox r w h hh = getBBox $ subterm1
      BBox r' w' h' hh' = getBBox $ subterm2
      nW = w + w' -- largura
      nR = ( ( r*(w/nW) + (w/nW + r'*(w'/nW)) ) / 2 ) -- posição da raiz
      nH = max h h' -- máxima altura dos filhos
      rWH = nW/nH -- relação largura/altura dos filhos
      dR = nW - (r*w) - (1-r')*w' -- distancia entre raizes
      nHH = max (0.1*dR*rWH) defh -- nova distancia raiz/filhos
  return $ AppB subterm1 subterm2 (BBox nR nW (nH+nHH) nHH)

```

O próximo passo é definir a função `renderTermB`, que será chamada pela interface para desenhar o termo lambda. Ela utiliza duas funções auxiliares: `rootPoint` para calcular exatamente os pontos do desenho correspondentes às raízes da subárvore, e `renderTermPB` que percorre o termo anotado realizando a renderização com base em um ponto de referência passado como parâmetro.

```

-- Calcula o ponto exato da raiz baseado nas dimensões da caixa de contorno
-- e onde a caixa se situa dentro do desenho (ponto superior esquerdo)
rootPoint :: BBox -> (Double,Double) -> (Double,Double)
rootPoint (BBox r w h h') (x,y) = (x + (r*(w)) , y)

-- Testa se o termo anotado é uma abstração
isLambdaB :: TermBBox -> Bool
isLambdaB (LambdaB _ _) = True
isLambdaB _ = False

-- Desenha o termo utilizando primitivas da biblioteca Cairo
renderTermB :: TermBBox -> Double -> (Double,Double) -> DrawingArea -> Render ()
renderTermB t z (a,b) da = do ctx <- liftIO $ widgetGetPangoContext da
  setLineWidth (max (0.5/z) 1.5) -- z experimental
  scale z z
  translate a b
  renderTermPB t (0,20) ctx

-- Função que executa a renderização do termo anotado recursivamente
renderTermPB :: TermBBox -> (Double,Double) -> PangoContext -> Render ()
renderTermPB (VarB s b) p ctx = do
  let rp = (rootPoint b p)

  -- cria o texto e obtém dimensões
  pL <- liftIO $ layoutText ctx s
  (_,PangoRectangle px py pw ph) <- liftIO $ layoutGetExtents pL

  -- desenha o fundo branco

```



```

setSourceRGB 1 1 1
rectangle (fst rp-(3+pw/2)) (snd rp-(3+ph/2)) (pw+6) (ph+6)
fill

-- desenha o texto
setSourceRGB 0 0 0
moveTo (fst rp-(pw/2)) (snd rp-(ph/2))
showLayout pL

renderTermPB (LambdaB s a bb1) p ctx = do
  let bb2 = getBBox a
      rp1 = rootPoint bb1 p
      rp2 = rootPoint bb2 ((\ (a,b)->(a,b+(bbrootheight bb1))) p)

  -- desenha linha
  setSourceRGB 0 0 1
  moveTo (fst rp1) (snd rp1)
  lineTo (fst rp2) (snd rp2)
  stroke

  -- cria o texto e obtém dimensões
  pL <- liftIO $ layoutText ctx $ "\"" ++ s
  (_,PangoRectangle px py pw ph) <- liftIO $ layoutGetExtents pL

  -- desenha o fundo branco
  setSourceRGB 1 1 1
  rectangle (fst rp1-(5+pw/2)) (snd rp1-(5+ph/2)) (pw+10) (ph+10)
  fill

  -- desenha o texto
  setSourceRGB 0 0.5 0
  moveTo (fst rp1-(pw/2)) (snd rp1-(ph/2))
  showLayout pL

  -- desenha subárvores
  renderTermPB a ((\ (a,b)->(a,b+(bbrootheight bb1))) p) ctx

renderTermPB (AppB a b bb1) p ctx = do
  let
    bb2 = getBBox a
    bb3 = getBBox b
    rp1 = rootPoint bb1 p
    rp2 = rootPoint bb2 ((\ (a,b)->(a,b+(bbrootheight bb1))) p)
    rp3 = rootPoint bb3 ((\ (a,b)->(a+(bbwidth bb2),b+(bbrootheight bb1))) p)

  -- desenha linha esquerda (possivelmente redex)
  if (isLambdaB a)
  then do
    setSourceRGB 1 0 1
    moveTo (fst rp1) (snd rp1)
    lineTo (fst rp2) (snd rp2)
    stroke
  else do
    setSourceRGB 0 0 1
    moveTo (fst rp1) (snd rp1)
    lineTo (fst rp2) (snd rp2)
    stroke

  -- desenha linha direita

```

```

setSourceRGB 0 0 1
moveTo (fst rp1) (snd rp1)
lineTo (fst rp3) (snd rp3)
stroke

-- cria o texto e obtém dimensões
pL <- liftIO $ layoutText ctx $ "@"
(,PangoRectangle px py pw ph) <- liftIO $ layoutGetExtents pL

-- desenha fundo branco
setSourceRGB 1 1 1
arc (fst rp1) (snd rp1) (max (5+pw/2) (5+ph/2)) 0 (2*pi)
fill

-- desenha texto
setSourceRGB 1 0 0
moveTo (fst rp1-(pw/2)) (snd rp1-(ph/2))
showLayout pL

-- desenha subárvores
renderTermPB a ((\ (a,b)->(a,b+(bbrootheight bb1))) p) ctx
renderTermPB b ((\ (a,b)->(a+(bbwidth bb2),b+(bbrootheight bb1))) p) ctx

```

10 Construções em Lambda Cálculo

Esta seção apresenta algumas codificações comuns de tipos de dados como termos lambda. A sintaxe das definições, com exceção dos numerais 0,1,2..., pode ser carregada diretamente no interpretador.

```

-- Combinadores básicos
i  = \x. x ;
k  = \x y. x;
s  = \x y z. x z (y z);

-- Combinador de ponto fixo
fix  = \f. (\x. f (x x)) (\x. f (x x)) ;

-- Lógica booleana
true  = \a b. a ;
false = \a b. b ;
if    = \c a b. c a b ;
and   = \a b. a b a ;
or    = \a b. a a b ;
not   = \p. p false true ;
xor   = \a b. (or (and (not a) b) (and a (not b)));
sss   = \a b. (not (xor a b));

-- Pares
pair  = \a b c. c a b ;
fst   = \p. p true ;
snd   = \p. p false ;

-- Números naturais
zero  = \f x. x ;
succ  = \n p. \q. p (n p q) ;
pred  = \n. fst (n (\p.(pair (snd p) (succ (snd p))))) (pair zero zero)) ;
add   = \m n p q. (m p) (n p q) ;
sub   = \m n. (n pred) m ;
isZero = \n. n (\x. false) true ;

```

```

equal    = \m n. and (isZero (sub m n)) (isZero (sub n m)) ;
lessEq   = \m n. isZero (sub m n) ;
great    = \m n. not ( isZero (sub m n)) ;
greatEq  = \m n. isZero (sub n m) ;
less     = \m n. not (isZero (sub n m)) ;
mult     = \m n p q. m (n p) q ;
div      = fix (\M. \m n. if (less m n) zero (succ (M (sub m n) n))) ;
mod      = fix (\M. \m n. if (less m n) m (M (sub m n) n)) ;
exp      = \m n. n m ;
fatorial = fix (\M. \n. if (isZero (pred n)) 1 (mult n (M (pred n)))) ;

-- Listas
empty    = \x. true ;
cons     = \h t. pair (pair false h) t ;
isEmpty  = \p. p (\x.\y.false) ;
head     = \l. snd (fst l) ;
tail     = \l. snd l ;
length   = fix (\M. \l. if (isEmpty l) 0 (succ (M (tail l)))) ;

```