λ-lifting

Robin Cockett

October 7, 2016

1 Introduction

Almost all programming languages have a technique of allowing "local definitions" : in Haskell this is provided by the "where" clause which attaches local definitions to a definition of larger scope. Here we use the "let" construct rather than the "where" construct – all this does is to change the order of where the declarations are made. An advantage of local definitions for the programmer is that he can use variables and names from an outer scopes while writing a local program and this provides for him an economy.

Unfortunately, the compiler for a functional language often has to unravel this economy so as to make all the local functions have global scope. This makes the program much easier to compile and obviates the necessity of having "activation records" on the stack. This makes calling functions – which in a functional language happen frequently – much more efficient. The process of moving all functions into global scope is called λ-lifting.

Here is an example of a program with local definitions using "let":

```
fun main x y = let
                   fun add p = add_to_x p
                   fun add_to_x q = (add_to_y q) + x
                   fun add_to_y q = q + y
                   in add y + x  end
```

The same function using "where" clauses is as follows:

```
fun main x y = add y + x   where
                   fun add p = add_to_x p
                   fun add_to_x q = (add_to_y q) + x
                   fun add_to_y q = q + y
```

After one has performed λ-lifting the program should look like:

```
fun main x y = add (x+y) x y
fun add p x y = add_to_x p x y
fun add_to_x q x y = (add_to_y q x y) +x
fun add_to_y q y = q+y
```

so that all functions are defined at the top-level. Notice how one has to add arguments to the functions in order that the "local" variable are still understood by the program. In this simple example it was quite straight forward to determine which arguments had to be added ... but it quickly becomes more complicated as one has recursive functions and nested "let" clauses.

2 α-renaming

The first problem to be surmounted in the process of λ-lifting is that there may be name clashes which will cause considerable problems as one moves functions into global scope. These name clashes can arise both from local variables and local function name clashes. Your very first task is therefore to ensure that all variable and function names are unique in the program: this is an α-conversion process.

Here is an example program in which there are multiple name clashes:

```
fun main x y z= let
    fun f y = x + g y ;
    fun g z = let
        fun f x = x * z
        in f x end
    in g z + f x end
```

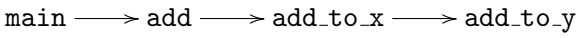
When one removes the name clashes one gets:

```
fun main x1 x2  x3 = let
    fun f1 x4 = x1 + f2 x4 ;
    fun f2 x5 = let
        fun f3 x6 = x6 * x5
        in f3 x1 end
    in f2 x3 + f1 x1 end
```

At this stage it is possible that a variable is free in the whole program (which should generate an error) or a function has the wrong number of arguments (which also should generate an error).

3 The call graph

Given a program an important structure is the "call graph": this tells you which functions call which other functions. Our first program above has a very simple call graph:



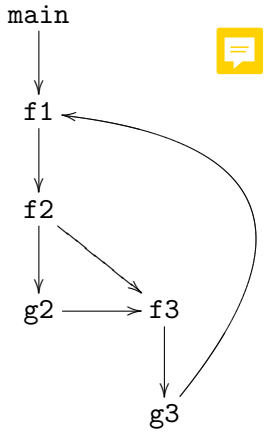
and it is because the call graph is simple that the λ-lifting problem is simple. Clearly if `main` calls **add** then `main` must have as argument all the variable which are used by `add`. This simple observation is the basis of how one works out how many variables it is necessary to add in order to have the function make sense at the top-level.

Here is a more complicated program (with no variable clashes):

```
fun main x y z n =
    let
        fun f1 v = x + f2 v
        fun f2 j = let
                       fun g2 b = b + f3 j
                   in  g2 y + f3 x end
```

```
        fun f3 k = let
                       fun g3 c  = c * f1 k
                   in  g3 z   end
    in   f1 n  end
```

and here is its call graph:



The basic idea of λ-lifting is that a function should have all the arguments of any function which it calls so that it can be moved into the top scope. In particular, one should make any variable that a function itself uses from an outer scope into one of its arguments. Thus, `f1` should have `x` as an argument as it uses `x`, but also, less obviously, `f1` also uses `z` ...

4 Performing λ lifting

After having removed all the variable clashes, by performing an α-conversion, one calculates the call graph. To each function in the call graph one associates two sets:

V_{args}	the arguments of the function
V_{free}	the free variable in the function body

Thus in this last example one has:

$$\begin{aligned}
\text{function} &: (V_{\mathsf{args}}, V_{\mathsf{free}}) \\
\texttt{main} &: (\{\texttt{x},\texttt{y},\texttt{z},\texttt{n}\},\{\texttt{n}\}) \\
\texttt{f1} &: (\{\texttt{v}\},\{\texttt{x},\texttt{v}\}) \\
\texttt{f2} &: (\{\texttt{j}\},\{\texttt{y},\texttt{x}\}) \\
\texttt{g2} &: (\{\texttt{b}\},\{\texttt{b},\texttt{j}\}) \\
\texttt{f3} &: (\{\texttt{k}\},\{\texttt{z}\}) \\
\texttt{g3} &: (\{\texttt{c}\},\{\texttt{c},\texttt{k}\})
\end{aligned}$$

The next step is to, for each link in the call graph, $f \longrightarrow g$ (i.e. whenever f calls g to ensure that the free variable of f include the variable which g uses but are not arguments. This can be expressed as an equation in which one is updating the free variables of f:

$$V'_{\mathsf{free}}(f) = V_{\mathsf{free}}(f) \cup (V_{\mathsf{free}}(g) \backslash V_{\mathsf{args}}(g))$$

One repeatedly updates the free variable sets associated to the links until no more variables are being added to the sets: this is called the fixed point. Here it is advantageous to start with the functions which are at the "bottom" of the call graph (in so far as cycles make this possible) so that their variable needs are passed up in one sweep.

In this example the first few updates are as follows:

function	initial	`f3` $\longrightarrow$ `g3`	`g2` $\longrightarrow$ `f3`	`f2` $\longrightarrow$ `g2`	`f1` $\longrightarrow$ `f2`
`main`	{x,y,z,n},{n}				
`f1`	{v},{x,v})				{v},{x,v,y,z}
`f2`	{j},{y,x}			{j},{y,x,j,z}	
`g2`	{b},{b,j}		{b},{b,j,z}		
`f3`	{k },{ z }	{k },{z,k}			
`g3`	{c},{c,k})				

Clearly the recursive call `g3` $\longrightarrow$ `f1` will have the effect of causing the eventual fixed point to be:

$$
\begin{aligned}
\text{function} &: (V_{\mathsf{args}}, V_{\mathsf{free}}) \\
\texttt{main} &: (\{\texttt{x},\texttt{y},\texttt{z},\texttt{n}\},\{\texttt{n},\texttt{x},\texttt{y},\texttt{z}\}) \\
\texttt{f1} &: (\{\texttt{v}\},\{\texttt{v},\texttt{x},\texttt{y},\texttt{z}\}) \\
\texttt{f2} &: (\{\texttt{j}\},\{\texttt{j},\texttt{x},\texttt{y},\texttt{z}\}) \\
\texttt{g2} &: (\{\texttt{b}\},\{\texttt{b},\texttt{j},\texttt{x},\texttt{y},\texttt{z}\}) \\
\texttt{f3} &: (\{\texttt{k}\},\{\texttt{k},\texttt{x},\texttt{y},\texttt{z}\}) \\
\texttt{g3} &: (\{\texttt{c}\},\{\texttt{c},\texttt{k},\texttt{x},\texttt{y},\texttt{z}\})
\end{aligned}
$$

The final step is to move all the functions to the top level and to call each function with all the variables in V_{free}. Thus, here is the result:

```
fun main x y z n =  f1 n x y z
fun f1 v x y z = x + (f2 v x y z)
fun f2 j x y z =  (g2 y j x y z) + (f3 x x y z)
fun g2 b j x y z = b + (f3 j x y z)
fun f3 k x y z = g3 z k x y z
fun g3 c k x y z = c * (f1 k x y z)
```

5 Data types for lifting

Your second assignment is to write a λ-lifting program for a programming language with arithmetic expressions, conditionals. function calls, and let expressions. You should work on the abstract syntax tree of the language to produce a modified abstract syntax tree with all the embedded functions lifted to the top-level. Here is the abstract syntax tree (you should use this!):

```
module AST where

--------------------------------------------------------------------------
```

```
-------------------------------------------------------------------------------
-- The data type for programs to lambda lift:
-------------------------------------------------------------------------------
-- Programs are lists of function declarations (the first in the list is
-- the main program.
-------------------------------------------------------------------------------

data Prog a b = Prog [Fun a b]

-------------------------------------------------------------------------------
--  Function declarations: first argument is function name (usually a string),
--  next is the arguments of the function (usually a list of strings)
--  finally there is the body of the function.
-------------------------------------------------------------------------------

data Fun a b = Fun (a,[b],Exp a b)

-------------------------------------------------------------------------------
-- There are two types of expressions: Boolean expressions and
-- arithmetic expressions.
-------------------------------------------------------------------------------
--  Boolean expressions allow <,>,== comparisons of arithmetic expressions
--  and logical operations &&,||,not.
-------------------------------------------------------------------------------

data BExp a b   = Lt (Exp a b) (Exp a b)
                | Gt (Exp a b) (Exp a b)
                | Eq (Exp a b) (Exp a b)
                | AND (BExp a b) (BExp a b)
                | OR (BExp a b) (BExp a b)
                | NOT (BExp a b)

-------------------------------------------------------------------------------
-- Arithmetic expressions allow +,*,/,- (binary and unary), constants,
-- variables, conditionals, function applications and let expressions.
-------------------------------------------------------------------------------

data Exp a b =  ADD (Exp a b) (Exp a b)
          | SUB (Exp a b) (Exp a b)
          | MUL (Exp a b) (Exp a b)
          | DIV (Exp a b) (Exp a b)
          | NEG (Exp a b)
          | CONST Int
          | VAR b
          | COND (BExp a b) (Exp a b)(Exp a b)
```

```
          | APP a [(Exp a b)]
          | LET [Fun a b] (Exp a b)
```

In the labs you will be provided with a parser (or indeed you can write your own) for these programs so that you can test your code easily.

Here is a basic pretty printer for the programs which attempts to lay out the program in a reasonable fashion. It will work for either integer or string variable and function names:

```
-------------------------------------------------------------------------
-- For pretty printing programs
-------------------------------------------------------------------------

-- Avoiding the behaviour of show on strings
-- (idea from stack overflow!) add instances as needed.  Can pretty print programs
-- with Stings and Int
-------------------------------------------------------------------------


class Printer a where
   printer:: a -> String

instance Printer a => Printer [a] where
   printer [] = []
   printer (a:as) = (printer a)++(printer as)

instance Printer Int where
   printer n = "v"++show (n::Int)

instance Printer Char where
   printer c = [c]

-- to print n spaces
spaces 0 = ""
spaces n = " "++(spaces (n-1))

-- pretty printing a program with basic indentation
show_prog:: (Printer a, Printer b) => (Prog a b) -> String
show_prog (Prog funs) = (concat (map (\f -> (show_fun 0 f)++"\n") funs))

show_fun:: (Printer a, Printer b) => Int -> (Fun a b) -> String
show_fun n (Fun (fname,a1:args,body)) = (spaces n) ++ "fun "++(printer fname)
```

```
          ++ "("++(printer a1)++(concat(map (\a ->","++(printer a)) args))
          ++") = "++ (show_exp' n body)
show_fun n (Fun (fname,[],body)) = (spaces n) ++ (printer fname) ++ "() = "
          ++ (show_exp' n body)

show_exp:: (Printer a, Printer b) => Int -> (Exp a b) -> String
show_exp n exp = (spaces n)++(show_exp' n exp)

show_exp':: (Printer a, Printer b) => Int -> (Exp a b) -> String
show_exp' n (ADD e1 e2) = (show_exp' n e1)++"+"++(show_exp' n e2)
show_exp' n (MUL e1 e2) = (show_exp' n e1)++"*"++(show_exp' n e2)
show_exp' n (DIV e1 e2) = (show_exp' n e1)++"/"++(show_exp' n e2)
show_exp' n (SUB e1 e2) = (show_exp' n e1)++"-"++(show_exp' n e2)
show_exp' n (NEG e) = "-"++(show_exp' n e)
show_exp' n (CONST m) = show m
show_exp' n (VAR b) = printer b
show_exp' n (COND b e1 e2) = "(if "++(show_bexp n b)++"\\n"
                    ++(spaces (n+3))++"then "++(show_exp' (n+3) e1)++"\n"
                    ++(spaces (n+3))++"else "++(show_exp' (n+3) e1)++")"
show_exp' n (APP f (e:es)) = (printer f)++"("++(show_exp' n e)
                    ++(concat(map (\x-> ","++(show_exp' n x)) es))++")"
show_exp' n (LET [] e) = (show_exp' n e)
show_exp' n (LET (f:fs) e) = "let\n"++(show_fun (n+3) f)
                    ++(concat (map (\f -> "\n"++(show_fun (n+3) f)) fs))
                    ++"\n"++(spaces n)++"in  "++(show_exp' n e)

show_bexp:: (Printer a, Printer b) => Int -> (BExp a b) -> String
show_bexp n (Lt e1 e2) = (show_exp' n e1)++"<"++(show_exp' n e2)
show_bexp n (Gt e1 e2) = (show_exp' n e1)++">"++(show_exp' n e2)
show_bexp n (Eq e1 e2) = (show_exp' n e1)++"=="++(show_exp' n e2)
show_bexp n (AND e1 e2) = (show_bexp n e1)++"&&"++(show_bexp n e2)
show_bexp n (OR e1 e2) = (show_bexp n e1)++"||"++(show_bexp n e2)
show_bexp n (NOT e) = "not("++(show_bexp n e)++")"

-------------------------------------------------------------------------
--        Tests
-------------------------------------------------------------------------

test1 = putStr (show_prog ((Prog
             [Fun ("main",[],(ADD (VAR "x") (VAR "y")))
             ,Fun ("f",["x"], (LET
                   [Fun ("g",["y"],MUL (VAR "y") (VAR "x"))
                   ,Fun ("h",["x","y"], DIV (VAR "x") (VAR "y"))]
                     (ADD (APP "g" [VAR "x"])
                     (APP "h" [VAR "x",CONST 7])) ))]) ::(Prog String String)))
```

```
test2 = putStr (show_prog ((Prog
              [Fun ("main",[],(ADD (VAR 1) (VAR 2)))
              ,Fun ("f",[1], (LET
                    [Fun ("g",[2],MUL (VAR 2) (VAR 1))
                    ,Fun ("h",[1,2], DIV (VAR 1) (VAR 2))]
                      (ADD (APP "g" [VAR 1])
                      (APP "h" [VAR 1,CONST 7])) ))]) ::(Prog String Int)))
```