Lambda Lifting Transformation

July 24, 2017

1 Lambda Lifting

Once pattern-matching compilation is completed, the next step in the interpretation of MPL programs is lambda lifting. MPL allows the programmers to define local functions. However, Core MPL doesn't allow for local function definitions. Thus all the local MPL functions must be put in the global scope. Lambda lifting is the transformation that puts the local functions in the global scope. The defn and where are the two MPL constructs that allow for local function definitions.

In this chapter, the difference between the defn and where constructs, and the strategy for lambda lifting of each contruct is discussed. An algorithm for lambda lifting was given Thomas Johnsson. The algorithm described in the chapter is an adaptation of Johnsson's algorithm customised for MPL.

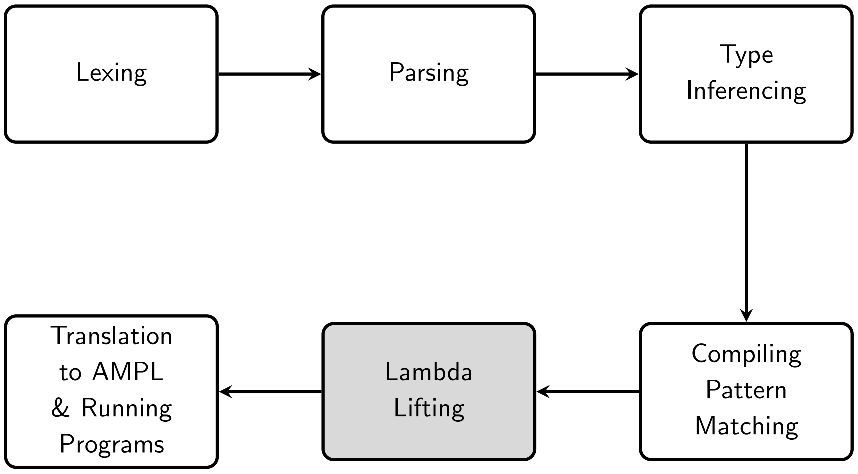
Lexing
Parsing
Type Inferencing
Compiling Pattern Matching
Lambda Lifting
Translation to AMPL & Running Programs

Figure 1: Interpretation Stages of MPL

2 Local Function Definitions with defn and where Constructs

Although both defn and where can be used to define local functions, there is a significant difference these constructs: the defn construct doesn't allow local variables while the main of the where construct is to allow non local variables. This has been described in Section 2.1 and Section 2.2 with examples.

2.1 defn Construct

The defn construct is used to modularize programs in MPL. The defn construct has the form:

```
defn
   definition_1
        ⋮
   definition_m
where
   definition_1
        ⋮
   definition_n
```

As can be seen in above, defn consists of two components:

- **Main body of defn**: MPL definitions between the keywords **defn** and **where** form the main body of defn. These definitions can be (co)data, (co)protocol, process, or function definitions. These definitions are visible to the entire MPL program.
- **where clause of defn**: **where** clause also allows for definitions. However, the scope of these defintions are different from the scope of definitions in the **main body** of the defn construct. The (co)data definitions in the **where** clause of defn are only visible to the other definitions in the **where** clause of that defn. The function definitions are visible to the other definitions in the **where** clause as well as the **main body** of that defn.

Table 1 shows an example of a pattern matching compiled MPL program that uses the defn construct. The program defines a **Tree** data structure and a **treeToBST** function. **treeToBST** takes a normal tree and returns a binary serach tree (BST) corresponding to the original binary tree. A BST is a binary tree with the following properties:

- All the elements in left subtree of any given node are smaller than the node element.
- All the elements in right subtree of any given node are greater than the node element
- Duplicate elements are not allowed.

The strategy used in the example in Table 1 to convert the binary tree to a binary search tree makes use of **List** data structure which is defined in the **where** clause of the defn construct. Program extracts the list of integers from the original tree. This list is then used to create a BST by inserting each element of list in a way that the BST property is never violated.

Function **insertBST** takes an integer and inserts it into a BST such that resultant tree is also a BST. Function **fromListToBST** folds over a list creating a BST with the elements of the list. Function **getIntList** folds over a binary tree of integers and gets the list of integers from the tree. **append** function appends two lists. Function **createBST** takes a tree of integers, gets the list of integers from the tree and subsequently returns a BST made out of the list of integers extracted from the input tree. All the functions mentioned in this paragraph have been defined **where** clause of the defn construct.

The function **treetoBST** defined in the **main body** of defn construct takes a binary tree and returns a BST by calling the **createBST** function defined in the **where clause** of the defn construct.

The **List** data type is visible only to the functions in the **where** clause of that defn and not to the definitions in the **main body** of defn or to any definitions in MPL outside the defn construct. The functions **append, insertBST, fromListToBST, getIntList,** and **createBST** are visible only to the other functions in the **where clause** or to the defintions in the **main body** of that defn.

This scoping scheme has the advantage that the function **treetoBST** doesn't need to know the underlying implementation of the **createBST** function which can be changed at anytime without impacting any other aspect of the MPL program.

In Section 3, the lambda lifting algorithm for the defn construct is discussed.

2.2 where Construct

where is a sequential MPL construct used to define local function that are needed only inside the body of a particular function. Table 2 shows an example of a pattern matching compiled MPL function that uses the where construct. The function **exFun** takes two numbers and a list of numbers as inputs. The outputs of **exFun** is the following:

- If the list of numbers is empty, then the first number is given as an output.
- For a list of non empty numbers, a sum is given as output. Every even number in the list contributes first argument of **exFun**, i.e **n1** to the sum. Every odd number constributes second argument **n2** to the sum

Function **exFun** has a local variable **ans** and two local functions **helpFun1** and **helpFun2** defined using the where construct. The local variable and local functions are visible only in either the function body or the body of the where construct itself.

An important thing to notice in the example in Table 2 is that both functions **helpFun1** and **helpFun2** have free variables. Free variables of a function are the set of variables used in the function body that are not the elements of the set of parameters of that function rather they are defined in a higher scope. Free variables for **helpFun1** and **helpfun2** are {n1} and {n1,n2} respectively.

Section 4 deals with the lambda lifting of local function definitions defined with where construct. The lambda lifting algorithm for where construct is different from that of the defn construct because local functions defined using where can have free variables where as those defined using defn can't have free variables.

Before one proceeds to lambda lifting, all the local variables introduced in the where clause (**ans** in Table 2) should be converted to constant functions (functions of arity 0). Once a local variable definition is converted to a constant function, to use this variable in the function body one needs to make a function call to the corresponding constant function. Program in Table 3 is obtained as a result of making locally defined variable **ans** a constant function.

```
defn
  data Tree(A) -> C = Leaf :: -> C
                      Node :: A,C,C -> C

  fun TreeToBST :: Tree(Int) -> Tree(Int) =
    tree -> createBST(tree)

where
  data List(A) -> C = Nil  ::     -> C
                      Cons :: A,C -> C

  -- append two lists
  append :: List(A),List(A) -> List(A) =
    t1,t2 -> fold t1 of
               Nil  :    -> t2
               Cons :x,r -> Cons(x,r)

  -- take an integer and inserts it into a Binary Search Tree
  fun insertBST :: Int,Tree(Int) -> Tree(Int) =
    elem,bst ->
      case bst of
        Leaf ->
          Node (elem,Leaf,Leaf)
        Node(n,t1,t2) ->
          case (n == elem) of
            True  ->
              bst
            False ->
              case (elem < n) of
                True ->
                  Node (n,insertBST (elem,t1),t2)
                False ->
                  Node (n,t1,insertBST(elem,t2))

  -- takes a list of integers and creates a binary
  -- search tree out of it
  fun fromListToBST :: List(Int) -> Tree(Int) =
    list -> fold list of
              Nil  :     -> Leaf
              Cons : x,r -> insertBST(x,r)

  -- get a list of integers from a tree of integers
  fun getIntList :: Tree(Int) -> List(Int) =
    tree -> fold tree of
              Leaf : -> Nil
              Node :x,ls1,ls2 -> append (Cons(x,ls1),ls2)

  -- take a simple binary tree and return a BST
  fun createBST :: Tree(Int) -> Tree(Int) =
    tree -> fromListToBST (getIntList (tree))
```

Table 1: Example : Local functions with defn Construct

```
fun exFun =
    n1,n2,list ->
         ans
      where
        ans = helperFun1 (list)

        fun helpFun1 =
            ls -> case ls of
                    Nil -> n1
                    Cons(x,xs) -> helperFun2(ls)

        fun helperFun2 =
            ls -> case ls of
                    Nil -> 0
                    Cons(y,ys) ->
                      if mod(y,2) == 0
                        then n1 + helperFun2(ys)
                        else n2 + helperFun2(ys)
```

Table 2: Example : Local Functions with where Construct

```
fun exFun =
    n1,n2,list ->
        ans()
      where
        fun ans =
               -> helperFun1 (list)

        fun helpFun1 =
            ls -> case ls of
                    Nil -> n1
                    Cons(x,xs) -> helperFun2(ls)

        fun helperFun2 =
            ls -> case ls of
                    Nil -> 0
                    Cons(y,ys) ->
                      if mod(y,2) == 0
                        then n1 + helperFun2(ys)
                        else n2 + helperFun2(ys)
```

Table 3: Example : Local Functions with where Construct

3 Lambda Lifting Local Functions of defn Construct

A naive approach to lambda lifting the local functions of defn construct would be to directly put all the definitions in the **where** clause of the defn construct and the definitions in **main body** of the defn construct, in the global scope. However, there is a problem with this approach as the names of the function definitions in the **where** clause of the defn construct may be the same as the names of the function definitions in the global scope. Thus, before the function definitions of the **where** clause of the defn construct are lifted to global scope, they must be renamed to an unique new global name.

Steps in lambda lifting the local functions of defn construct are be listed as below:

- Rename the function names of the function definitions in the where clause to unique new global names. Once functions are renamed, the function calls made with old names should be altered to reflect the name change.
- The definitions in the **main body** and the **where** clause of defn can now be lifted to the outermost scope.

The steps have been applied to function definitions in the **where** clause of the defn construct in the program in Table 1. The steps of lambda lifting have been shown in Table 4 and 5. Table 4 shows the step where the local functions in the **where clause** of the defn construct is renamed to unique global names. Table 5 shows the step where the renamed functions in the **where clause** of the defn construct are lifted to the outermost scope.

4 Lambda Lifting Local Functions of where Construct

If one tries lifting the local functions of the where, the free variables in the local functions of the where construct will no longer be in scope. The lambda lifting algorithm for the where construct is thus more complicated than that for the defn construct because of the presence of non-local variables in the function definitions of the where construct.

Lambda lifting algorithm for the local functions of the where construct consists of the following steps:

1. **Rename functions and their parameters**: All the function names of the local function definitions in the where clause are renamed uniquely to ensure that the local functions have a unique name. This is done because before lambda lifting transformation is performed, different functions in the different scopes may have the same name without any ambiguity. However, once lambda lifting transformation is performed they will be in the same scope and thus need unique names. Care should be taken to convert the function calls made with old function names to function calls with new function names.

   The arguments of all the function definitions are also renamed with fresh variables to ensure that every function has unique parameters. This step is required for the third step of the algorithm.

2. **Compute Set Equations for functions**: For every local function **f** defined in the *where* construct, the following triple is computed.

   $$(\{\textbf{free vars}\}, \{\textbf{bound vars}\}, \{\textbf{funs used}\})$$

   - **free vars** are the free variables of function **f**.
   - **bound vars** are the arguments of function **f**.
   - **funs used** are the functions called inside **f**.

```
defn
  data Tree(A) -> C = Leaf :: -> C
                      Node :: A,C,C -> C

  fun TreeToBST :: Tree(Int) -> Tree(Int) =
    tree -> unqFun5(tree)

where
  data List(A) -> C = Nil  ::     -> C
                      Cons :: A,C -> C

  -- append two lists
  fun unqFun1 :: List(A),List(A) -> List(A) =
    t1,t2 -> fold t1 of
               Nil  :    -> t2
               Cons :x,r -> Cons(x,r)

  -- take an integer and inserts it into a Binary Search Tree
  fun unqFun2 :: Int,Tree(Int) -> Tree(Int) =
    elem,bst ->
      case bst of
        Leaf ->
          Node (elem,Leaf,Leaf)
        Node(n,t1,t2) ->
          case (n == elem) of
            True  ->
              bst
            False ->
              case (elem < n) of
                True ->
                  Node (n,unqFun2 (elem,t1),t2)
                False ->
                  Node (n,t1,unqFun2 (elem,t2))

  -- takes a list of integers and creates a binary
  -- search tree out of it
  fun unqFun3 :: List(Int) -> Tree(Int) =
    list -> fold list of
              Nil  :     -> Leaf
              Cons : x,r -> unqFun2 (x,r)

  -- get a list of integers from a tree of integers
  fun unqFun4 :: Tree(Int) -> List(Int) =
    tree -> fold tree of
              Leaf : -> Nil
              Node :x,ls1,ls2 -> unqFun1 (Cons(x,ls1),ls2)

  -- take a simple binary tree and return a BST
  fun unqFun5 :: Tree(Int) -> Tree(Int) =
    tree -> unqFun3 (unqFun4 (tree))
```

Table 4: Step 1 : Rename local functions of the defn construct

```
data List(A) -> C = Nil  ::     -> C
                    Cons :: A,C -> C

data Tree(A) -> C = Leaf :: -> C
                Node :: A,C,C -> C

-- append two lists
fun unqFun1 :: List(A),List(A) -> List(A) =
  t1,t2 -> fold t1 of
             Nil  :    -> t2
             Cons :x,r -> Cons(x,r)

-- take an integer and inserts it into a Binary Search Tree
fun unqFun2 :: Int,Tree(Int) -> Tree(Int) =
  elem,bst ->
    case bst of
      Leaf ->
        Node (elem,Leaf,Leaf)
      Node(n,t1,t2) ->
        case (n == elem) of
          True  ->
            bst
          False ->
            case (elem < n) of
              True ->
                Node (n,unqFun2 (elem,t1),t2)
              False ->
                Node (n,t1,unqFun2 (elem,t2))

-- takes a list of integers and creates a binary
-- search tree out of it
fun unqFun3 :: List(Int) -> Tree(Int) =
  list -> fold list of
            Nil  :     -> Leaf
            Cons : x,r -> unqFun2 (x,r)

-- get a list of integers from a tree of integers
fun unqFun4 :: Tree(Int) -> List(Int) =
  tree -> fold tree of
            Leaf : -> Nil
            Node :x,ls1,ls2 -> unqFun1 (Cons(x,ls1),ls2)

-- take a simple binary tree and return a BST
fun unqFun5 :: Tree(Int) -> Tree(Int) =
  tree -> unqFun3 (unqFun4 (tree))


fun TreeToBST :: Tree(Int) -> Tree(Int) =
  tree -> unqFun5(tree)
```

Table 5: Step 2 : Lift the definitions of defn to the global scope

The pair of a function name and the triple for that function is called a **set equation** of the function.

$$\textbf{Set Equation}(\mathbf{f}) = (\mathbf{f}, (\{\textbf{free vars}\}, \{\textbf{bound vars}\}, \{\textbf{bfuns used}\}))$$

A set of **set equations** are obtained as a result of this step.

3. **Solve the Set Equations using a fixed point calculation**: The aim of this step is to generate the set of all the free variables for a function **f**. This set not only contains the free variables that are directly present in the body of the function **f** but also the set of free variables present in the body of functions called inside **f**.

   Once the set of set equations corresponding to the local functions defined in the where is generated, the equations are then solved to get the all the free variables for these functions. The set equations are solved using a fixed point calculation.

   The fixed point of a function **p** is a value **x**, such that

$$\mathbf{p\ \ x\ =\ x}$$

   The algorithm for solving the set equations is discussed in Figure 2. The function **solveEqns** takes a set of set equations and returns back a set of solved set equations. Solved set equation of a function symbol **f** is the set equation that contains the set of all the free variables corresponding to **f**.

   Function **solveEqn** in Figure 2 solves a set equation with respect to the original set of set equations. **solveEqn** iteratively changes the set equation it is trying to solve until the fixed point condition for that set equation is reached. The fixed point condition for a set equation is that the pair of the set equation being solved and the original set of set equations don't change after an iteration. The free variable set associated with the set equation of function **f** when the fixed point condition is reached is the set of all the free variables for **f**.

   During every iteration in solving of the set equations for function symbol **f**, the set of free variables and the set of function calls made inside **f** are changed. These are respectively the first and third element in the tuple associated with function symbol **f** in the set equation.

   Function **fixFun** in Figure 2 does the iterative change to the set equation of **f** by performing the following actions for every function **g** which is a member of third element of the tuple associated with **f**:

   - Lookup **g** and find its associated tuple of free variables, bound variables and function calls.
   - The modified free variable set for **f** is the union of sets of free variables for **f** and **g** with the set of bound variables of **f** removed from the union.
   - The modified function call set is the union of the function call set of **f** and **g** with the singleton set $\{g\}$ removed from the union.
   - The bound variable set remains unmodified.

4. **Add the free variables to their corresponding functions**: Once all the free variables corresponding to a function are generated, they are added to the parameters of the function

```
solveEqns :: $\{Set\ \ Eqn\}\ -> \ \{Set\ \ Eqn\}$
solveEqns es =
    for each s in es do
      solveEqn (s,es)


solveEqn :: $(Set\ \ Eqn, \{Set\ \ Eqn\})\ -> \ Set\ \ Eqn$
solveEqn ((f,(fv,bv,fl)),es) =
    ((f,(fv',bv',fl')),es') <- fixFun ((f,(fv,bv,fl)),es)
    case fv' == fv of   -- fixed point condition (fv doesn't change)
      True  -> (f,(fv',bv',fl'))
      False ->solveEqn ((f,(fv',bv',fl')),es')


fixFun :: $(Set\ \ Eqn, \{Set\ \ Eqn\})\ -> \ (Set\ \ Eqn, \{Set\ \ Eqn\})$
fixFun ((f,(fv,bv,fl)),es) =
  for every g in fl do
    (g,(gfv,gbv,gl)) <- lookup g in es
    let
      fv' = ((fv $\cup$ gfv) $\setminus$ bv)
      fl' = ((fl $\cup$ gfl) $\setminus$ g)
    output: ((f,(fv',bv,fl')),es)
```

Figure 2: Solving Set Equations using Fixed Point Calculation

definition. This ensures that there are no free variables in the function body. Since the arity of the function has changed, the arguments of the function call must be expanded with the free variables of that function.

5. **Push the local functions to global scope**: Since there are no free variables in the local functions any more they can be pushed to the global scope. This is the final step in the lambda lifting transformation.

The stepwise lambda lifting transformation for the local function in the where construct of function **exFun** defined in Table 2 is shown in Tables 6 and 7.

Original Program

```
fun exFun =
  n1,n2,list ->
      ans()
    where
      fun ans =
             -> helperFun1 (list)

      fun helperFun1 =
          ls -> case ls of
                  Nil -> n1
                  Cons(x,xs) -> helperFun2(ls)

      fun helperFun2 =
          ls ->
            case ls of
              Nil -> 0
              Cons(y,ys) ->
                if mod(y,2) == 0
                  then n1 + helperFun2(ys)
                  else n2 + helperFun2(ys)
```

Step 1 : Renamed function names and parameters

```
fun exFun =
  n1,n2,list ->
      unq_fn1()
    where
      fun unq_fn1 =
          -> unq_fn2(list)

      fun unq_fn2 =
        u1 -> case ls of
                Nil -> n1
                Cons(x,xs) -> unq_fn3(u1)

      fun unq_fn3 =
        u2 ->
          case ls of
            Nil -> 0
            Cons(u3,u4) ->
              if mod(u3,2) == 0
                then n1 + unq_fn3(u4)
                else n2 + unq_fn3(u4)
```

Step 2: Generate Set Equations for Local Functions

```
SetEqn(unq_fun1) = (unq_fun1,({list},{},[unq_fun2]))
SetEqn(unq_fun2) = (unq_fun2,({n1},{u1},[unq_fun3]))
SetEqn(unq_fun3) = (unq_fun3,({n1,n2},{u2},[unq_fun3]))
```

Table 6: Lambda Lifting *exFun* (Continued on Table 7)

Step 3: Solve Set Equations

The solved set equations are as follows:

```
SetEqn(unq_fun1) = (unq_fun1,($\{list, n1, n2\}$,{},[unq_fun2]))
SetEqn(unq_fun2) = (unq_fun2,($\{n1, n2\}$,$\{u1\}$,[unq_fun3]))
SetEqn(unq_fun3) = (unq_fun3,($\{n1, n2\}$,$\{u2\}$,[unq_fun3]))
```

Thus, the free variables for unq_fun1 are $\{list, n1, n2\}$.
For function unq_fun2 and unq_fun3 are $\{n1, n2\}$.

Step 4: Augment free variables to function parameters and call

```
fun exFun =
  n1,n2,list ->
      helperFun1 (list,n1,n2)
    where
      fun unq_fn1 =
        u1 -> case ls of
                Nil -> n1
                Cons(x,xs) -> unq_fn2(u1,n1,n2)

      fun unq_fn2 =
        u2 ->
          case ls of
            Nil -> 0
            Cons(u3,u4) ->
              if mod(u3,2) == 0
                then n1 + unq_fn2(u4,n1,n2)
                else n2 + unq_fn2(u4,n1,n2)
```

Step 5: Lift Local Functions to Outermost Scope

```
fun unq_fn2 =
  u2 ->
    case ls of
      Nil -> 0
      Cons(u3,u4) ->
        if mod(u3,2) == 0
          then n1 + unq_fn2(u4,n1,n2)
          else n2 + unq_fn2(u4,n1,n2)

fun unq_fn1 =
  u1 -> case ls of
          Nil -> n1
          Cons(x,xs) -> unq_fn2(u1,n1,n2)

fun exFun =
  n1,n2,list -> unq_fn1(list,n1,n2)
```

Table 7: Lambda Lifting *exFun* (Continued from Table 6)