

Lambda Lifting Transformation

July 13, 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 construct 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.

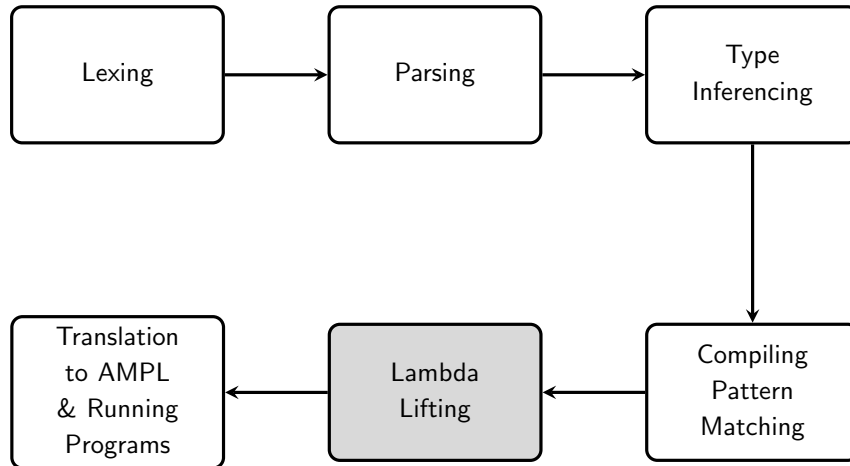


Figure 1: Interpretation Stages of MPL

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. Table 1 shows an example of a pattern matching compiled MPL program that uses the **defn** construct. The program takes two lists and returns back a list where each element of the final list is the sum of the corresponding elements of the original list. In case of lists of unequal length, the sum of the corresponding elements is calculated till the elements in the smaller list is exhausted and the remainder of the elements in the longer list are ignored.

As can be seen in Table 1, **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:** There may be some definitions that are required only for the definitions in the main body of the **defn** construct. These definitions are written in the **where** clause of the **defn** construct. Scope of a definition in the **where** clause of **defn** is:
 - other definitions in the where clause, and
 - the definitions in the main body of the **defn** construct.

For example, in Table 1, the **List** data type is in the scope of functions **helperfun**, and **zipSum** and not in scope of **mainFun**. Similarly, function **helperFun** is in the scope of function **zipSum** and not in the scope of **mainFun**. Function **zipSum** is in scope of function **mainFun**. Thus MPL uses **defn** construct to implement data hiding and to control the visibility of definitions.

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

```
fun mainFun =
  x,y -> zipSum(x,y)

defn
  fun zipSum =
    x,y -> helperFun (x,y)
  where
    data List(A) -> C = Nil  ::      -> C
                      Cons :: A,C -> C

    fun helperFun
      l1,l2 ->
      case l1 of
        Nil -> Nil
        Cons(x,xs) -> case l2 of
          Nil -> Nil
          Cons(y,ys) -> Cons((x+y),helperFun(xs,ys))
```

Table 1: Example : Local functions with **defn** Construct

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 contributes 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.

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

Table 2: Example : Local Functions with where Construct

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**

```

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)

```

Table 3: Example : Local Functions with where Construct

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.

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 a 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. These steps have been shown in Table 4.

Rename local function helperFun of defn to uniqNm1
<pre> fun mainFun = x,y -> zipSum(x,y) defn of fun zipSum = x,y -> uniqName1(x,y) where data List(A) -> C = Nil :: -> C Cons :: A,C -> C fun uniqName1 l1,l2 -> case l1 of Nil -> Nil Cons(x,xs) -> case l2 of Nil -> Nil Cons(y,ys) -> Cons((x+y),uniqName1(xs,ys)) </pre>
Push the definitions of defn to the global scope
<pre> data List(A) -> C = Nil :: -> C Cons :: A,C -> C fun uniqName1 l1,l2 -> case l1 of Nil -> Nil Cons(x,xs) -> case l2 of Nil -> Nil Cons(y,ys) -> Cons((x+y),uniqName1(xs,ys)) fun zipSum = x,y -> uniqName1(x,y) fun mainFun = x,y -> zipSum(x,y) </pre>

Table 4: Lambda lifting Local functions defined with **defn** construct

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.

$$(\{\mathbf{free\ vars}\}, \{\mathbf{bound\ vars}\}, \{\mathbf{funcs\ used}\})$$

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

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

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

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 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 5 and 6.

```

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,gfl)) <- lookup g in es
  let
    fv' = ((fv ∪ gfv) \ bv)
    fl' = ((fl ∪ gfl) \ g)
  output: ((f,(fv',bv,fl')),es)

```

Figure 2: Solving Set Equations using Fixed Point Calculation

Original Program
<pre> 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) </pre>
Step 1 : Renamed function names and parameters
<pre> 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) </pre>
Step 2: Generate Set Equations for Local Functions
<pre> 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])) </pre>

Table 5: Lambda Lifting *exFun* (Continued on Table 6)

Step 3: Solve Set Equations
<p>The solved set equations are as follows:</p> $\begin{aligned} \text{SetEqn}(\text{unq_fun1}) &= (\text{unq_fun1}, (\{list, n1, n2\}, \{\}, [\text{unq_fun2}])) \\ \text{SetEqn}(\text{unq_fun2}) &= (\text{unq_fun2}, (\{n1, n2\}, \{u1\}, [\text{unq_fun3}])) \\ \text{SetEqn}(\text{unq_fun3}) &= (\text{unq_fun3}, (\{n1, n2\}, \{u2\}, [\text{unq_fun3}])) \end{aligned}$ <p>Thus, the free variables for unq_fun1 are $\{list, n1, n2\}$. For function unq_fun2 and unq_fun3 are $\{n1, n2\}$.</p>
Step 4: Augment free variables to function parameters and call
<pre> 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) </pre>
Step 5: Lift Local Functions to Outermost Scope
<pre> 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) </pre>

Table 6: Lambda Lifting *exFun* (Continued from Table 5)