Lambda Lifting Transformation

July 7, 2017

1 Lambda Lifting

Once pattern-matching compilation is completed, the next step in the compilation of MPL programs is lambda lifting transformation. MPL allows the programmers to define local functions. However, CMPL doesn't allow for local function defintions and thus all the local MPL functions should be put in the global scope. Lambda Lifting is the transformation that puts the local functions in the global scope.

2 Examples of Local Functions in MPL

The defn and let are the two MPL constructs that allow for local function defintions. The difference between the local functions defined with defn and let is that the functions defined with let can have *free variables* where as the ones defined with *defn* can't have *free variables*. *free variables* of a function are the variables used inside a function definition that are not elements of the set of that function's parameters.

Examples highlighting the difference between the local functions defined using *defn* and let have been given in Table 1. The local function definition *myFun* in the *where* clause of defn statement doesn't have any free variables. However, functions *xFun* and *yFun* in the *where* clause of let statement have free variables n_1 and n_2 respectively.

3 Lambda Lifting in MPL

As seen earlier *lambda lifting* is the transformation that puts the local functions in the global scope. The complexity of the transformation arises from the presence of *free variables* in the function definitions. If a function has free variables in its definition, then putting it in global scope will make the free variables out of scope which would result in a semantic error. Example - This can be seen in the local functions $xFun$ $yFun$ defined with let in the Table 1 having n_1 and n_2 as free variables respectively. If $xFun$ and $yFun$ were to be moved to the same scope level as *rFun* then variables n_1 and n_2 will become out scope in their respective functions.

Therefore, before a function *f* is placed in the global scope, it should be ensured that it doesn't contain any *free variables*. One way to do this can be to find all the *free variables* of *f* and append them to the function parameters of f. Now all the free variables of *f* become bound and they can be placed in the global scope. This technique is viable except for one flaw. The *free variables* of the functions called inside *f* also contribute to the free variables of *f*. Thus, the *free variables* obtained from these functions called inside *f* should be unioned with the *free variables* directly present in *f*.

Local Function with defn (no free variables)	Local Functions with let (has free variables n_1 and n_2)

<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>
	<pre>fun rFun =
	    n1, n2, nl ->
	      let
	        xFun(nl)
	      where
	        fun xFun =
	          l ->
	            case l of
	              [] -> n1
	              (x:xs) -> x + yFun (xs)
	
	        fun yFun =
	          l ->
	            case l of
	              [] -> n2
	              (x:xs) -> x + yFun (xs)</pre>

Table 1: Local functions in MPL

The *free variables* calculated as a result of this can be added to the parameter this and the inner scope functions can be *lambda lifted*.

For example - The free variable of function *yFun* and *xFun* in Table 1 can be calculated as follows.

$$
\begin{aligned}
freevars(yFun) &= \{n_2\} \ \cup \ freevars(yFun) \\
&= \{n_2\}
\end{aligned}
$$

$$
\begin{aligned}
freevars(xFun) &= \{n_1\} \ \cup \ freevars(yFun) \\
&= \{n_1, \ n_2\}
\end{aligned}
$$

4 Lambda Lifting Algorithm

This algorithm is used to *lambda lifts* functions that can have free variables in their definitions. In the case of function defintions in the let statement that is the case, therefore this algorithm would be required to *lambda lift* the function. However, in the case of *defn* there are no free variables in the local functions and thus the function can directly be lambda lifted after renaming (the step1 and step 5 of the algorithm).

Lambda lifting consists of the following steps :-

1. **Rename functions and their parameters** - All the function names are renamed uniquely to ensure that they have an unique name. This is done because before *lambda lifting*, differ-

ent functions in different scopes may have the same name without ambiguity. However, once lambda lifted they will be in the same scope and thus need unique names.

The parameters of all the function defintions are also renamed with fresh variables to esnure that every function has unique parameters.

2. **Compute Set Equations for Functions** - For every function *f* defined inside the *where* clause of the let statement, the following triple is computed.

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

- *free vars* are the *free variables* in f.
- *bound vars* are parameters of f.
- *funs used* are the list of functions called made inside f.

The pair of the function name and the triple for that function is called a *Set Equation* of the function.

$$Set\ Equation(f) = (f, (\{free\ vars\}, \{bound\ vars\}, [funs\ used]))$$

A list of *Set Equations* are obtained as a result of this step.

3. **Solving the Set Equations** - Once the list of *Set Equations* is generated, the equations are then solved to get the *free variables* for all the functions defined inside the *where* clause of the let statement.

```
for every set equation (f,(fv,bv,fl)) in set equation list sl do
    for every function name n in fl do
        if fl == [] or fl == [f]
           then
             return (f,fv)
           else
            (nfv,nbv,nfl)  <- lookup n in sl
            modify the set equation to (f,((fv ∪ nfv) \ bv), bv, nfl \n)
```

4. **Add the free variables to their corresponding functions** - Once 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.

5 Examples of Lambda Lifting MPL programs

The first example *lambda lifting* transformation is demonstrated (in Table 2) on the program defined with the *defn* construct from Table 1. Since, there are no *free variables* present in the local functions in the *where* clause of *defn*, the local functions can directly be pushed to the global scope once the function and argument renaming has been completed.

Second example demonstrates *lambda lifting* local functions defined using `let` statement in Table 3. The program defined using `let` from Table 1 has been chosen to demonstrate this.

```
Original Program :

    fun last =
      l -> case l of
       (x:[]) -> x
       (x:xs) -> last(xs)

    defn of
        fun someFun =
            x, y -> myFun (x,y)
    where
      fun myFun  =
         l1,l2 ->
           case l1 of
             []     -> []
             (x:xs) -> case l2 of
                         [] -> []
                         (y:ys) -> (x+y):myFun (xs,ys)
```

```
Renamed functions and parameters :

    fun fn1 =
      u1 -> case u1 of
              (x:[]) -> x
              (x:xs) -> last(xs)

    defn of
        fun fn2 =
            u2, u3 -> fn3 (u2,u3)
    where
      fun fn3  =
         u4,u5 ->
           case u4 of
             []     -> []
             (x:xs) -> case u5 of
                         [] -> []
                         (y:ys) -> (x+y):myFun (xs,ys)
```

```
Lambda Lifted Program :

    fun fn1 =
      u1 -> case u1 of
              (x:[]) -> x
              (x:xs) -> last(xs)

    fun fn2 =
        u2, u3 -> fn3 (u2,u3)

    fun fn3  =
         u4,u5 ->
           case u4 of
             []     -> []
             (x:xs) -> case u5 of
                         [] -> []
                         (y:ys) -> (x+y):myFun (xs,ys)
```

Table 2: Lambda lifting Local functions defined with *defn* construct

```
Original Program

fun rFun =
    n1, n2, nl ->
      let
        xFun(nl)
      where
        fun xFun =
          l ->
            case l of
              []     -> n1
              (x:xs) -> x + yFun (xs)

        fun yFun =
          l ->
            case l of
              []     ->  n2
              (x:xs) -> x + yFun (xs)
```

```
Renamed functions and parameters :

fun fn1 =
    u1, u2, u3 ->
      let
        fn2 (u3)
      where
        fun fn2 =
          u4 ->
            case u4 of
              []      -> u1
              (u5:u6) -> u5 + fn3 (u6)

        fun fn3 =
          u7 ->
            case u7 of
              []     ->  u2
              (u8:u9) -> u8 + fn3 (u9)
```

```
Using the free vars of fn2 and fn3
to augment the parameter and arguments
of the respective functions.

freeVars (fn2) = {u1,u2}
freeVars (fn3) = {u2}

fun fn1 =
    u1, u2, u3 ->
      let
        fn2 (u3,u1,u2)
      where
        fun fn2 =
          u4 u1 u2 ->
            case u4 of
              []      -> u1
              (u5:u6) -> u5 + fn3 (u6,u2)

        fun fn3 =
          u7, u2 ->
            case u7 of
              []     ->  u2
              (u8:u9) -> u8 + fn3 (u9,u2)
```

```
Lambda Lifted Program



fun fn1 =
    u1, u2, u3 ->
       fn2 (u3,u1,u2)

fun fn2 =
  u4 u1 u2 ->
    case u4 of
      []      -> u1
      (u5:u6) -> u5 + fn3 (u6,u2)

fun fn3 =
  u7, u2 ->
    case u7 of
      []     ->  u2
      (u8:u9) -> u8 + fn3 (u9,u2)
```

Table 3: Local functions defined with let construct