Compilation of Pattern Matching

April 20, 2017

1 Compilation of MPL Programs to Core MPL

Once an MPL program is processed by the frontend, i.e the lexer and the parser, an AST is generated which is a faithful representation of the original program. This AST is used to type infer the MPL programs.

The next step in compilation of MPL programs is to compile *pattern-matching* to the *case* statements and *lambda lifting* the MPL programs which involves making all the local functions global. *Core MPL(CMPL)* is an intermediate language of MPL which gets rid of the syntactic sugar from MPL programs and represents every program in terms of a reduced set of language constructs.

2 Example of Pattern-Matching Compilation

Pattern-matching is a syntactic feature of MPL which allows the programmer to write readable programs such as the following :-

```
fun myZip =
     [],   []   -> []
     [],   ys   -> []
     xs,   []   -> []
     x:xs, y:ys -> x:append (xs,ys)
```

In the above example, the constructors on the left hand side of the fucntion defintion (*Nil and Cons*) are called *patterns*. In the CMPL stage, the patterns are compiled into *case* statements. For example, CMPL representation of the *myZip* function will look like the following.

```
fun myZip =
    xl,yl ->
        case xl of
          [] ->
            case yl of
               []   -> []
               y:ys -> []
          (x:xs) ->
            case yl of
              []   -> []
              y:ys -> <x,y>:myZip (xs,ys)
```

3 Steps of Pattern-Matching Compilation

The compilation from *pattern-matching* to the *case* statement can be done in following steps :-

1. Take a function that uses *pattern-matching* and represent it with the *Match* construct. The Match construct is described in section 3.1.

2. Reduce the *Match* construct repeatedly until the base case is reached and reduction is not possible anymore. The reduction of the *Match* construct has been described in the section 3.2.

3. Once the base case is reached, the *pattern-matching* is successfully compiled to case statement. Handling the base case has been described in section 3.3

3.1 Match Construct

Match construct can be described as the following Haskell data type :-

```
data Match    = Match [String] [Equation] Term

type Equation = ([Pattern],Term)
```

The first argument of the *Match* constructor is a list of variable names. The number of elements in this list is the same as the number of patterns in any given line of *pattern-matching*. It should be noted that every well formed function definition that uses *pattern-matching* should have same number of patterns in every *pattern-matching* line. The second argument of *Match* is an equation list. Every equation in the list captures a line of *pattern-matching*. The last argument is a term acts like a default term while dealing with missing constructors of a data type in *pattern-matching*.

An example of a function represented with a *Match* construct is as below.

```
fun myFun =
    a,[],     ys     -> P a ys
    a,(x:xs), []     -> Q a xs
    a,(x:xs), (y:ys) -> R x xs y ys
```

An AST representation of the above function can be following :-

```
    Fun ‘‘myFun’’
            [
             ([
               VarPatt ‘‘a’’,
               ConsPatt (‘‘Nil’’,[]),
               VarPatt ‘‘ys’’
              ],
                    TCons (‘‘P’’,[TVar ‘‘a’’,TVar ‘‘ys’’])),
             ([
               VarPatt ‘‘a’’,
               ConsPatt (‘‘Cons’’,[VarPatt ‘‘x’’,VarPatt ‘‘xs’’]),
               ConsPatt (‘‘Nil’’,[])
               ],
                    TCons (‘‘Q’’,[TVar ‘‘a’’,TVar ‘‘xs’’])),
```

```
            ([
              VarPatt ``a'',
              ConsPatt (``Cons'',[VarPatt ``x'',VarPatt ``xs'']),
              ConsPatt (``Cons'',[VarPatt ``y'',VarPatt ``ys''])
             ],
                   TCons (``R'',[TVar ``x'',TVar ``xs'',TVar ``y'',TVar ``ys''])
            )
           ]
```

VarPatt is used to represent variable patterns. It takes an argument of type String, which is the variable that it is representing. *ConsPatt* is used to represent constructor patterns. This takes as argument a pair. First element of the pair is the constructor name and second element is the list of arguments that this constructor takes as input. For example -

```
ConsPatt (``Cons'',[VarPatt ``x'',VarPatt ``xs''])
```

Three constructors P, Q and R have been used as terms in the definition of *myFun*.

Like *ConsPatt* represents constructor patterns, *TCons* represents constructor terms. *TCons* takes a pair as argument. The first element of the pair is the name of the constructor and second element is the list of terms that this constructor takes as input.

For example -

```
TCons (``S'',[TVar ``x'',TVar ``xs'',TVar ``y'',TVar ``ys''])
```

This represents a term constructor S taking four terms as inputs.

Given an AST for the function, the corresponding *Match* representation of the function can be easily generated. The *Match* representation for the body of *myFun* function is given below. Note that instead of representing *Match* with the heavy data type representation like the AST, a more readable and succinct representation has been used.

```
fnMatch = Match [u1,u2,u3]
            [
              ([a,Nil,ys],               P a ys),
              ([a,Cons(x,xs),Nil],       Q a xs),
              ([a,Cons(x,xs),Cons(y,ys)],R x xs y ys)
            ]
            t'
```

In the *Match* construct above t' is the default term to be used with constructors of a data type that are missing from the *pattern-matching*. It can be an error message or an exception. The final function definition for *myFun* with compiled pattern matching can be obtained as follows.

```
fun myFun =
    u1,u2,u3 -> reduced fmMatch
```

The reduction of the *Match* construct has been discussed in the next section (3.2).

3.2 Reducing the Match Construct

Depending on whether the first pattern of the *pattern matching* starts with a variable, a constructor or a mixture of variables and constructors, there are three reduction rules.

1. Variable Rule
2. Constructor Rule
3. Mixture Rule

3.2.1 Variable Rule

This reduction rule is used when every equation in *Match* starts with a *variable pattern*. The reduction rule has been captured in Table 1 below. $t_i[M/x]$ means that in term t_i, M has been substituted for x.

Before	After
Match (u:us) [(v_1:vs_1,t_1), . . . (v_n:vs_n,t_n)] t′	Match us [(vs_1,t_1[u/v_1]), . . . (vs_n,t_n[u/v_n])] t′

Table 1: Reduction Rule for Variable Case

An example of using the *Variable Rule* for reducing a *Match* construct has been shown below in Table 2. The *Match* construct in the before column in the table below is that of function *myFun* from section 2.

Before	After
Match [u_1,u_2,u_3] [([a,Nil,ys], P a ys), ([a,Cons(x,xs),Nil], Q a xs), ([a,Cons(x,xs),Cons(y,ys)],R x xs y ys)] t′	Match [u_2,u_3] [([Nil,ys], P u_1 ys), ([Cons(x,xs),Nil], Q u_1 xs), ([Cons(x,xs),Cons(y,ys)],R x xs y ys)] t′

Table 2: Example for Variable Case

3.2.2 Constructor Rule

This reduction rule is used when every equation in *Match* starts with a *constructor pattern*. The reduction rule has been captured in Table 3 below.

There can be mutiple equations in the *Match* construct starting with the same constructor name. The equations with the same constructor name are grouped together by interchanging the order of equations (to which these *constructor patterns* belong) in the *Match* construct.

It should be noted that order of two consecutive equations with constructor patterns in the beginning can be interchanged only when the constructor names in these patterns are different.

The *Match* construct with the grouped equations, will have the below structure where every equation $\mathbf{eq_i}$ is the list of equations starting with the same constructor $\mathbf{C_i}$. ++ is the *append* function.

```
Match (u:us)
        [
          eq_1 ++ ... ++ eq_k
        ]
        t'
```

Consider that constructor $\mathbf{C_i}$ for equation $\mathbf{eq_i}$ takes $\mathbf{r}$ arguments, $\mathbf{C_i\ b_1\ \ldots\ b_r}$. For succintness, instead of using $\mathbf{C_i\ b_1\ \ldots\ b_r}$, the representation $\mathbf{C_i\ bs}$ is used where $\mathbf{bs}$ represents the list of arguments that the constructor $\mathbf{C_i}$ takes. i.e $\mathbf{bs} = [\mathbf{b_1}, \ldots, \mathbf{b_r}]$.

Consider that $\mathbf{eq_i}$ has j equations each starting with a constructor $\mathbf{C_i}$. $\mathbf{eq_i}$ can then be represented as below.**rs** represents the remaining patterns of the equation,i.e the ones except the first constructor pattern.

```
eq_i = [
         ( ((C_i bs_i1):rs_i1),t_i1),
                .
                .
                .
         ( ((C_i bs_ij):rs_ij),t_ij)
       ]
```

Before	After

`Match (u:us)`
`[`
`  eq_1 ++ ... ++ eq_k`
`]`
`t'`
	`case u of`
	`C_1 ns_1 -> Match (ns_1 ++ us) eq'_1 t'`
	`.  .`
	`.  .`
	`.  .`
	`C_k ns_k -> Match (ns_k ++ us) eq'_k t'`

Table 3: Reduction Rule for Constructor Case

In the *After* column of Table 3, $\mathbf{ns_i}$ is the list of fresh variables of length equal to the number of arguments that the constructor $\mathbf{C_i}$ takes. Every $\mathbf{eq'_i}$ has the following form.

```
eq'_i = [
          (bs_i1 ++ rs_i1, t_i1),
                .
                .
                .
          (bs_ij ++ rs_ij, t_ij)
        ]
```

An example of using the *Constructor Rule* for reducing a *Match* construct has been shown below in Table 4. The *Match* construct in the before column in the table below is the *Match* construct from the *After* column of Table 2 with its equations partitioned according to the constructor names in the first pattern.

Before	After

Match [u_2,u_3]
  (
  [([Nil,ys],P u_1 ys)] ++
  [([Cons(x,xs),Nil], Q u_1 xs),
  ([Cons(x,xs),Cons(y,ys)],R x xs y ys)]
)
  t′
	case u_2 of
	  Nil ->
	  Match [u_3] [([ys], P u_1 ys)] t′
	
	  Cons u_4 u_5 ->
	  Match [u_4,u_5,u_3]
	  [
	  ([x,xs,Nil], Q u_1 xs),
	  ([x,xs,Cons(y,ys)],R x xs y ys)
	  ]
	  t′

Table 4: Example for Constructor Case

3.2.3 Mixture Rule

This reduction rule is used when some equations in the *Match* construct starts with a *constructor pattern* and some with a *variable pattern*. The reduction rule has been captured in Table 5 below.

In the *Before* column of Table 5, the equations have been divided into k parts, $eq_1, \ldots, eq_k$. Every partition contains all the consecutive equations that begin either with a *variable pattern* or *constructor pattern*. For example - Consider the below *Match* construct.

```
Match [u_1,u_2]
    [
     ([Nil,ys],       A ys),
     ([xs,Nil],       B xs),
     ([Cons(x,xs),ys],C x xs ys)
```

Before	After
Match us [eq_1 ++ ... ++ eq_k] t′	Match us eq1 (Match us eq2 (... (Match us eq_k t')))

Table 5: Reduction Rule for Mixture Case

]
t′

Match construct with partitioned equations is as below.

```
Match [u1,u2]
  (
    [([Nil,ys],       A ys)] ++
    [([xs,Nil],       B xs)] ++
    [([Cons(x,xs),ys],C x xs ys)]
  )
  t′
```

Reduction of this *Match* construct using the mixture rule has been shown in the Table 6 below.

Before	After
Match [u_1,u_2] ([([Nil,ys], A ys)] ++ [([xs,Nil], B xs)] ++ [([Cons(x,xs),ys],C x xs ys)]) t′	Match [u_1,u_2] [([Nil,ys],A ys)] (Match [u_1,u_2] [([xs,Nil],B xs)] (Match [u_1,u_2] [([Cons(x,xs),ys],C x xs ys)] t′))

Table 6: Example for Mixture Case

3.3 Handling the Base Case

The base case is reached when the first argument of the *Match* construct which is a variable list, is empty. The base case can be divided into two cases.

1. When the equation list associated with the *Match* construct is *non-empty*. Note that in this case, the pattern list associated with every equation of the equation list will be empty.

```
Match []
     [
      ([],t1),
         .
         .
         .
      ([],tn)
     ]
     t'
```

2. When the equation list associated with the *Match* construct is *empty*

```
Match [] [] t'
```

Reduction of the base case *Match* constructs has been given in Table 7 below.

Before	**After**

`Match []`
`[`
`([],t1),`
`.`
`.`
`.`
`([],tn)`
`]`
`t'`
	`t1`

`Match [] [] t'`	`t'`

Table 7: Reduction Rule for Base Case

4 An Example of Pattern-Matching Compilation

In this section, reduction rules described in sections 3.2 and 3.3 are used to compile the *Match* construct for *myFun* function defined in section 3.1. The first two steps of the reductions have already been done as examples of using the *variable case* and *constructor case* in Table 2 and Table 4 respectively. The first *Match* construct in the table below is the *Match* construct from the *After* column of Table 4.

The function definition of *myFun* with compiled *pattern-matching* is as below :-

Step 1 :-
 	
case u_2 of
  Nil ->
  Match [u_3] [([ys], P u_1 ys)] t′
  Cons u_4 u_5 ->
  Match [u_4,u_5,u_3]
  [
  ([x,xs,Nil], Q u_1 xs),
  ([x,xs,Cons(y,ys)],R x xs y ys)
  ]
  t′
	Step 2 :-
	
	case u_2 of
	  Nil ->
	  Match [] [([], P u_1 u_3)]
	  Cons u_4 u_5 ->
	  Match [u_5,u_3]
	  [
	  ([xs,Nil], Q u_1 xs),
	  ([xs,Cons(y,ys)],R u_4 xs y ys)
	  ]
	  t′

Step 3 :-
 	
case u_2 of
  Nil ->
  P u_1 u_3
  Cons u_4,u_5 ->
  Match [u_3]
  (
  [([Nil], Q u_1 u_5)] ++
  [([Cons(y,ys)],R u_4 u_5 y ys)]
)
  t′
	Step 4 :-
	
	case u_2 of
	  Nil -> P u_1 u_3
	  Cons u_4,u_5 ->
	  case u3 of
	  Nil -> Q u_1 u_5
	  Cons u_6 u_7 ->
	  Match [u_6, u_7]
	  ([y,ys],R u_4 u_5 y ys)
	  t’

Step 5 :-
 	
case u_2 of
  Nil -> P u_1 u_3
  Cons u_4,u_5 ->
  case u3 of
  Nil -> Q u_1 u_5
  Cons u_6 u_7 ->
  Match []
  ([],R u_4 u_5 u_6 u_7)
  t’
	Step 6 :-
	
	case u_2 of
	  Nil -> P u_1 u_3
	  Cons u_4 u_5 ->
	  case u3 of
	  Nil -> Q u_1 u_5
	  Cons u_6 u_7 -> R u_4 u_5 u_6 u_7

Table 8: Reduction of Match for myFun

```
fun myFun =
   u1,u2,u3 ->
        case u2 of
          Nil -> P u1 u3
          Cons u4 u5 ->
            case u3 of
              Nil -> Q u1 u5
              Cons u6 u7 -> R u4 u5 u6 u7
```

5 Totality of MPL Functions

Currently every MPL function is required to be total for successful *pattern-matching* compilation. Functions that are not total will throw an error in the compilation owing to the missing constructors.