Compilation of Pattern-Matching

July 24, 2017

1 Compilation of MPL Programs to Core MPL

Once an MPL program is processed by the lexer and the parser, an Abstract Syntax Tree (AST) is generated which is a faithful representation of the original program. This AST is used for the type inferenece of the MPL program. If the program type checks, the next step in the compilation of MPL programs is to convert this AST to a simpler core language with a reduced set of language constructs, called Core MPL (CMPL).

The compilation of the AST to the CMPL happens in 2 steps. These steps are listed below in the order they are performed:

- **Pattern-Matching Compilation** - This steps translates the pattern-matching syntactic sugar that MPL programs have.
- **Lambda Lifting** - MPL allows the programmers' to define local functions. However, CMPL doesn't allow for local function defintions. Lambda Lifting transformation gets rid of local function defintions in MPL programs and makes all the local function global.

In this chapter, the **pattern-matching compiler algorithm** is discussed.

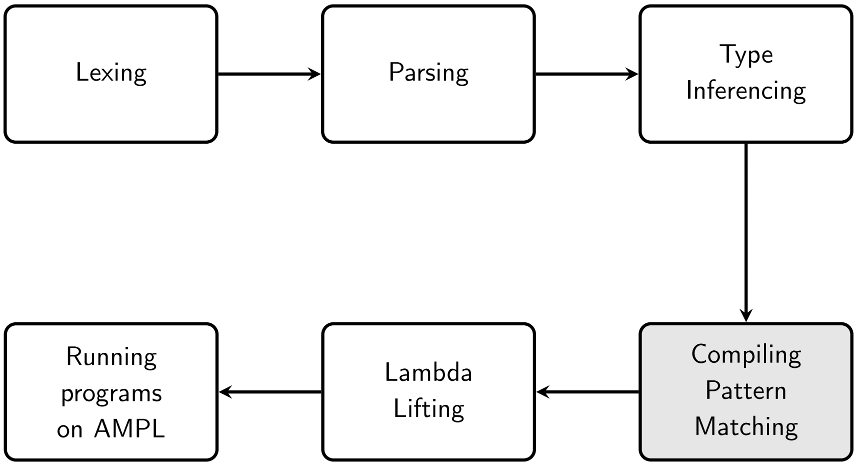

Figure 1: Compilation Stages : Pattern Matching Compilation

2 Pattern Matching in MPL

MPL allows the programmer to define readable functions using its the pattern-matching syntax. Figure 2 shows a function **f** defined with **m** lines of **n**-tuples of patterns and a sequential term. A line **i** consisting of the pair of patterns $\mathbf{p_{i,1}}, \ldots, \mathbf{p_{i,n}}$ and term $\mathbf{t_i}$ is known as a **pattern phrase**.

$$\begin{array}{l} fun\ \ f = \\ \quad p_{1,1},\ \cdots\ ,\ p_{1,n}\ \rightarrow\ t_1 \\ \qquad\qquad \cdots \\ \quad p_{m,1},\ \cdots\ ,\ p_{m,n}\ \rightarrow\ t_m \end{array}$$

Figure 2: Function Defintion with Pattern Matching

A pattern $\mathbf{p_{i,j}}$ can be one of the below patterns:

- Variable Pattern
- Constructor Pattern
- Record Pattern
- Product Pattern
- Don't Care Pattern represented by an unserscore ("_")

Don't care patterns can be thought as a special case of variable patterns in which the variables in the patterns of the pattern phrase are not used in the term of the pattern phrase. In the further discussion of pattern matching compilation, the don't care pattern is not explicitly discussed as its compilation scheme is the same as that of variable patterns.

A function can have a mixture of these patterns in any pattern phrase. The constructor, record and product patterns may have other patterns in their body. A pattern that contains another pattern is called a nested pattern.

The pattern matching compilation algorithm gets rid of the constructor, product and record patterns (including the nested patterns) in a function body. The final result of pattern matching compilation is a function body that contains only one pattern phrase consisting of just variable patterns.

Table 1 provides examples of functions defined with patterns. It also shows the pattern matching compiled form of these example functions.

append function definition in Table 1 appends two lists. It uses a mixture of constructor pattern (first argument) and variable pattern (second argument).

pairFun function takes two input arguments, a pair of list of integers and a pair of integers. Its output is the following:

- When both the lists of the first argument are empty, the output is the integer pair in the second argument.

- When the first list is non empty and second list is empty, the output is a pair. The first element of this pair is the head of the first list of the first argument of the function. The second element of the pair is the second element of the second parameter of the function.
- The case when the first list is empty and the second list is non empty is symmetric to the last case.
- If both the lists of the first pair are non empty then the two elements of the output pair are the head of the first and the second list respectively.

Function **fstOfHead** takes an infinite list of lists and tries to return the first element of the first list of the infinite list of lists. If the first list of the infinite list of lists is empty, then the function returns a failure with **FF** constructor else returns the first element of the first list wrapped in the **SS** constructor.

3 Overview of Pattern Matching Compiler Algorithm

In this section, the algorithm for step wise compilation of patterns in a function body is described. The pattern matching compilation algorithm is described with the help of **compile** function. **compile** function helps in a clean and concise exposition of the algorithm. The basic steps of the algorithm can be summarised as follows:

- Represent the body of a function definition (which has patterns) with the **compile** function. Section **??** describes the **compile** function in details.
- Define reduction rules for the **compile** function. A reduction rule is a syntactic rule to transform the **compile** function. There are mutiple reduction rules associated with the different configuration of **compile**. However, for any given configuration there is a unique reduction rule. The configurations of the **compile** function and their associated reduction rules have been described in Section **??**.
- Apply the reduction rules repeatedly till the normal form of the **compile** function is achieved. The normal form of **compile** function is a configuration of **compile** such that it can't be reduced any further. The normal form and the corresponding configuration of the **compile** function has been described in Section .

The normal form of **compile** will produce a new function body where mutiple pattern phrases of variable patterns, constructor patterns, record patterns, product patterns or nested patterns are compiled to a pair of string and a sequential term.

The pattern matching compiled form of function **f** defined in Figure 2 looks like:

$$\begin{aligned} &fun \;\; f = \\ &\quad u_1, \ldots, u_n \;\rightarrow \textbf{normal} \;\; \textbf{form} \;\; (compile \;\; function \;\; for \;\; f's \;\; body) \end{aligned}$$

Function defintion with pattern matching	**Function definition with compiled patterns**

fun append :: [A],[A] -> [A] =
  [],   ys   -> ys
  x:xs, ys   -> x:append (xs,ys)
	fun append :: [A],[A] -> [A] =
	  xl,yl ->
	    case xl of
	      Nil -> yl
	      Cons(x,xs) -> Cons(x,append(xs,yl))

fun pairFun::([Int],[Int]),(Int,Int) ->
             (Int,Int) =
  ([] ,[]),(p,q)   -> (p,q)
  (x:xs,[]),(p,q)   -> (x,q)
  ([],y:ys),(p,q)   -> (p,y)
  (x:xs,y:ys),(p,q) -> (x,y)
	fun pairFun::([Int],[Int]),(Int,Int) ->
	             (Int,Int) =
	  u,v ->
	    case π_0(u) of
	      Nil ->
	        case π_1(u) of
	          Nil -> (π_0(v),π_1(v))
	          Cons(y,ys) -> (π_0(v),y)
	      Cons(x,xs) ->
	        case yls of
	          Nil -> (x,π_1(v))
	          Cons(y,ys) -> (x,y)

codata C -> InfList(A)
          = Head :: C -> A
            Tail :: C -> C
 	
data SF(A) -> C = FF ::   -> C
                 SS :: A -> C
 	
fun fstOfHead::InfList([A]) -> SF(A) =
  (: Head := [] ,Tail := t :) -> FF
  (: Head := a:as,Tail := t :) -> SS(a)
	codata C -> InfList(A)
	          = Head :: C -> A
	            Tail :: C -> C
	
	data SF(A) -> C = FF :: -> C
	                 SS :: A -> C
	
	fun fstOfHead :: InfList([A]) -> SF(A) =
	  u -> case Head(u) of
	         Nil -> FF
	         Cons(x,xs) -> SF(x)

Table 1: Examples of Pattern Matching Compilation

4 compile Function

The **compile** function has the form:

$$\begin{array}{ll} \textit{compile} & [u_1, u_2, \ldots, u_n] \\ & [\,[\\ & \quad (\ [\ p_{1,1},\ \ \ldots\ ,\ p_{1,n}\ ],\ \ t_1) \\ & \quad (\ [\ p_{2,1},\ \ \ldots\ ,\ p_{2,n}\ ],\ \ t_2) \\ & \qquad\qquad \vdots \\ & \quad (\ [\ p_{m,1},\ \ \ldots\ ,\ p_{m,n}\ ],\ t_m) \\ &]\,] \\ & default\ \ term \end{array}$$

The type of the **compile** function is:

$$\textit{compile}\ \ ::\ \ [Seq.\ Term], [[(Pattern, Seq.\ Term)]], Seq.\ Term\ \ \rightarrow\ \ ([String], Seq.\ Term)$$

As can be above in the structure and the type of the **compile** function, it takes three inputs and outputs a pair of list of string and a sequential term. The input arguments to **compile** have been described below:

- First argument is a list of sequential terms. To begin with this list of terms consists of a list of fresh variables equal in number to the number of patterns in a pattern phrase.
- Second argument is a list of list of pattern phrases.
- Third argument is a sequential term. It is the default term to be used with the missing constructors of a data type in pattern-matching. It is also used in the reduction rule **Mixture Rule** described in Section 5.5.

Table 2 gives an example of the **compile** function correponding to the body of **append** function that contains pattern matching. Since the number of patterns in a pattern phrase in the **append** function is two, the first argument of the **compile** function is a list of variable terms containing fresh variables **u1** and **u2**.

Second argument of the **compile** function in Table Pmatch:ExampleCompile is the pattern phrase. In the representation of the pattern phrase here, constructor **VPatt** is used to represent a variable pattern. For example - **VPatt ys** represents the variable pattern **ys**. Similarly constructor **CPatt** is used to represent a constructor pattern. For example - **CPatt Cons** (**VPatt x**, **VPatt xs**) represents a constructor pattern **Cons** that takes two variable patterns **VPatt x** and **VPatt xs**.

Similarly, constructors **RPatt** and **PPatt** are used to represent record pattern and product pattern in this chapter.

5 Reduction Rules for compile Function

Following sections describe the reduction rules for the **compile** function. The reduction rule applicable to a particular **compile** function depends on what the first pattern of the pattern phrases (which is the second argument of the **compile**) is. These reduction rules are repeatedly applied to the **compile** function until the patterns associated with the pattern phrases is empty.

Before	**After**

`append :: [A],[A] -> [A] =`
`    [],  ys -> ys`
`    x:xs,ys -> x : append(xs,ys)`
	$compile\ \ [u_1, u_2]$
	$\Big[\Big[$
	$\Big(\Big[\ CPatt\ \ Nil\ ,\ VPatt\ \ ys\ \Big],\ ys\ \Big),$
	$\vdots$
	$\Big(\Big[\ CPatt\ \ Cons\ (VPatt\ \ x,\ VPatt\ \ xs), VPatt\ \ ys\ \Big],$
	$x : append(xs, ys)\Big)$
	$\Big]\Big]$
	$default\ \ term$

Table 2: Example : compile Function

5.1 Variable Rule

This rule is applicable if the first pattern of all the pattern phrases are variable patterns (represented by the constructor **VPatt**). This rule has been described in Table 3. In Table 3, the syntax $\mathbf{VPatt\ \ v_1}$ @ $\mathbf{p_{1,1}}$ represents that the first pattern of the first pattern phrase $\mathbf{p_{1,1}}$ has the structure of a variable pattern comprised of variable $\mathbf{v_1}$.

In the reduction rule in Table 3, the first variable pattern of every pattern phrase is substituted by the first term from the list of terms, the first argument of the **compile** function, inside the corresponding term of every pattern phrase. In the rule, the notation $\mathbf{t_i}\ [\mathbf{v_i}/\mathbf{u_i}]$ represents that the variable $\mathbf{v_i}$ has been substituted by a term $\mathbf{u_i}$ inside the body of term $\mathbf{t_i}$.

5.2 Constructor Rule

This rule is applicable if the first pattern of all the pattern phrases are constructor patterns (represented by the constructor **CPatt** that takes two arguments: first argument is the constructor name and second argument is the arguments that the constructor mentioned in the first argument takes).

Let us define a data type **D** with **e** number of constructors, $\mathbf{C_1}, \ldots, \mathbf{C_e}$ using which the constructor reduction rule in Table 4 is explained. The data definition of **D** has the form:

$$\begin{array}{lcl} data\ \ D\ \ (a_1..a_f) & = & C_1\ \ us_v \\ & & \quad\ldots \\ & & C_e\ \ us_z \end{array}$$

The constructor $\mathbf{C_1}$ in the defintion of **D** takes **v** number of arguments represented by $\mathbf{us_v}$.

A few important things about the Constructor Rule described in Table 4 are:

Before	**After**
$compile\ \ [\mathbf{u_1}, \ldots, u_n]$ $\Big[\Big[$ $\Big(\Big[\,(\mathbf{VPatt}\ \ \mathbf{v_1})\ @\ \mathbf{p_{1,1}},\ p_{1,2}, \ldots, p_{1,n}\,\Big], t_1\Big),$ $\vdots$ $\Big(\Big[\,(\mathbf{VPatt}\ \ \mathbf{v_m})\ @\ \mathbf{p_{m,1}},\ p_{m,2}, \ldots, p_{m,n}\,\Big], t_m\Big)$ $\Big]\Big]$ $default\ \ term$	$compile\ \ [u_2, \ldots, u_n]$ $\Big[\Big[$ $\Big(\Big[\,p_{1,2},\ \ldots\ ,\ p_{1,n}\,\Big], \mathbf{t_1}\ [\mathbf{v_1}/\mathbf{u_1}]\Big),$ $\vdots$ $\Big(\Big[\,p_{m,2},\ \ldots\ ,\ p_{m,n}\,\Big], \mathbf{t_m}\ [\mathbf{v_m}/\mathbf{u_1}]\Big)$ $\Big]\Big]$ $default\ \ term$

Table 3: Variable Rule

- In the before column, the syntax $\mathbf{CPatt}\ \ \mathbf{C_c}\ \big(\mathbf{p_{1,1,1}}, \ldots, \mathbf{p_{1,1,x}}\big)@\mathbf{p_{1,1}}$ represents that the first pattern of the first pattern phrase $\mathbf{p_{1,1}}$ has the structure of a constructor pattern comprised of constructor $\mathbf{C_c}$ that takes $\mathbf{x}$ arguments which are $(\mathbf{p_{1,1,1}}, \ldots, \mathbf{p_{1,1,x}})$. $\mathbf{p_{1,1,i}}$ represents that it is the $\mathbf{i^{th}}$ nested pattern of the first pattern of the first pattern phrase.

- The list of list of pattern phrases are partitioned such that all the pattern phrases consisting of a particular constructor in the first pattern are grouped together in a list.

- The after column of the rule shows that a case statement is generated casing on the different constructors of data type $\mathbf{D}$.

- Every constructor in the generated case takes fresh variables as arguments and generates a corresponding **compile** function. This **compile** function appends the terms corresponding to the fresh variable list to the front of the list $[\mathbf{u_2}, \ldots, \mathbf{u_n}]$. It also adds the inner patterns of the constructor to the front of respective pattern phrases.

5.3 Record Rule

5.4 Product Rule

5.5 Mixture Rule

5.6 Empty Rule and Normal Form

Before	**After**
$\textit{compile}\ \ [\mathbf{u_1}, \ldots, u_n]$ $\big[\ \big[$ $\Big(\Big[\mathbf{CPatt\ \ C_c}\ (\mathbf{p_{1,1,1}}, \ldots, \mathbf{p_{1,1,x}})@\mathbf{p_{1,1}}, \ldots, p_{1,n}\Big], t_1\Big),$ $\vdots$ $\Big(\Big[\mathbf{CPatt\ \ C_d}\ (\mathbf{p_{m,1,1}}, \ldots, \mathbf{p_{m,1,y}})@\mathbf{p_{m,1}}, \ldots, p_{m,n}\Big], t_m\Big)$ $\big]\ \big]$ $\textit{default\ \ term}$ **The above compile can be rewritten to:** $\textit{compile}\ \ [\mathbf{u_1}, \ldots, u_n]$ $\big[$ $\big[$ $\Big(\Big[\mathbf{CPatt\ \ C_1}\ (\mathbf{p_{g,1,1}}, \ldots, \mathbf{p_{1,1,v}})@\mathbf{p_{g,1}}, \ldots, p_{g,n}\Big], t_g\Big),$ $\vdots$ $\Big(\Big[\mathbf{CPatt\ \ C_1}\ (\mathbf{p_{h,1,1}}, \ldots, \mathbf{p_{h,1,v}})@\mathbf{p_{h,1}}, \ldots, p_{h,n}\Big], t_h\Big)$ $\big]$ $\vdots$ $\big[$ $\Big(\Big[\mathbf{CPatt\ \ C_e}\ (\mathbf{p_{i,1,1}}, \ldots, \mathbf{p_{i,1,z}})@\mathbf{p_{i,1}}, \ldots, p_{i,n}\Big], t_i\Big),$ $\vdots$ $\Big(\Big[\mathbf{CPatt\ \ C_e}\ (\mathbf{p_{j,1,1}}, \ldots, \mathbf{p_{j,1,z}})@\mathbf{p_{j,1}}, \ldots, p_{j,n}\Big], t_j\Big)$ $\big]$ $\big]$ $\textit{default\ \ term}$	**case** $\mathbf{u_1}$ **of** $\mathbf{C_1}\ \ (\mathbf{fv_1}, \ldots, \mathbf{fv_v})\ \ \rightarrow$ $\textit{compile}\ \Big([\mathbf{fv_1}, \ldots, \mathbf{fv_v}]\ {+}{+}\ [u_2 \ldots, u_n]\Big)$ $\big[\ \big[$ $\Big(\Big[\mathbf{p_{g,1,1}}, \ldots, \mathbf{p_{g,1,v}}, p_{g,2}, \ldots, p_{g,n}\Big], t_g\Big),$ $\vdots$ $\Big(\Big[\mathbf{p_{h,1,1}}, \ldots, \mathbf{p_{h,1,v}}, p_{h,2}, \ldots, p_{h,n}\Big], t_h\Big)$ $\big]\ \big]$ $\textit{default\ \ term}$ $\vdots$ $\vdots$ $\mathbf{C_e}\ \ (\mathbf{fv_1}, \ldots, \mathbf{fv_z})\ \ \rightarrow$ $\textit{compile}\ \Big([\mathbf{fv_1}, \ldots, \mathbf{fv_z}]\ {+}{+}\ [u_2 \ldots, u_n]\Big)$ $\big[\ \big[$ $\Big(\Big[\mathbf{p_{i,1,1}}, \ldots, \mathbf{p_{i,1,z}}, p_{i,2}, \ldots, p_{i,n}\Big], t_i\Big),$ $\vdots$ $\Big(\Big[\mathbf{p_{j,1,1}}, \ldots, \mathbf{p_{j,1,z}}, p_{h,2}, \ldots, p_{j,n}\Big], t_j\Big)$ $\big]\ \big]$ $\textit{default\ \ term}$

Table 4: Constructor Rule

Before	**After**
$\mathit{compile}\ \ [u_1,\ldots,u_n]$ $[\ [$ $\Big(\Big[\mathbf{RPatt}\ (\mathbf{D_1} : \mathbf{p_{1,1,1}},\ldots,\mathbf{D_k} : \mathbf{p_{1,1,k}})@\mathbf{p_{1,1}},\ldots,p_{1,n}\Big],t_1\Big),$ $\vdots$ $\Big(\Big[\mathbf{RPatt}\ (\mathbf{D_1} : \mathbf{p_{m,1,1}},\ldots,\mathbf{D_k} : \mathbf{p_{m,1,k}})@\mathbf{p_{m,1}},\ldots,p_{m,n}\Big],t_m\Big)$ $]\ ]$ $default\ \ term$	$\mathit{compile}\ \Big([\mathbf{D_1}\ \mathbf{u_1},\ldots,\mathbf{D_k}\ \mathbf{u_1}]\ \ ++$ $[u_2,\ldots,u_n]\Big)$ $[\ [$ $\Big(\Big[\mathbf{p_{1,1,1}},\ldots,\mathbf{p_{1,1,k}},p_{1,2},\ldots,p_{1,n}\Big],t_1\Big),$ $\vdots$ $\Big(\Big[\mathbf{p_{m,1,1}},\ldots,\mathbf{p_{m,1,k}},p_{m,2},\ldots,p_{m,n}\Big],t_m\ \Big)$ $]\ ]$ $default\ \ term$

Table 5: Record Rule

Before	**After**
$\mathit{compile}\ \ [u_1,\ldots,u_n]$ $[\ [$ $\Big(\Big[\mathbf{PPatt}\ (\mathbf{p_{1,1,1}},\ldots,\mathbf{p_{1,1,k}})\ @\ \mathbf{p_{1,1}},p_{1,2}\ldots,p_{1,n}\Big],t_1\Big),$ $\vdots$ $\Big(\Big[\mathbf{PPatt}\ (\mathbf{p_{m,1,1}},\ldots,\mathbf{p_{m,1,k}})\ @\ \mathbf{p_{m,1}},p_{m,2}\ldots,p_{m,n}\Big],t_m\Big)$ $]\ ]$ $default\ \ term$	$\mathit{compile}\ \Big([\pi_{\mathbf{0}}\ \mathbf{u_1},\ldots,\pi_{\mathbf{k-1}}\ \mathbf{u_1}]\ ++\ [u_2,\ldots,u_n]\Big)$ $[\ [$ $\Big(\Big[\mathbf{p_{1,1,1}},\ldots,\mathbf{p_{1,1,k}},p_{1,2},\ldots,p_{1,n}\Big],t_1\Big),$ $\vdots$ $\Big(\Big[\mathbf{p_{m,1,1}},\ldots,\mathbf{p_{m,1,k}},p_{m,2},\ldots,p_{m,n}\Big],t_m\ \Big)$ $]\ ]$ $default\ \ term$

Table 6: Product Rule

Before	**After**
$compile\ \ [u_1,\ldots,u_n]$ $\Big[\ \Big[$ $\Big(\Big[\mathbf{VPatt\ \ v_1@\ p_{1,1}},\ldots,p_{1,n}\Big],t_1\Big),$ $\vdots$ $\Big(\Big[\mathbf{VPatt\ \ v_q@\ p_{q,1}},\ldots,p_{q,n}\Big],t_q\Big),$ $\Big(\Big[\mathbf{CPatt\ \ C_1\ us}_{\text{q+1}}\mathbf{@\ p}_{\text{q+1},\mathbf{1}},\ldots,p_{\text{q+1},n}\Big],t_{\text{q+1}}\Big),$ $\vdots$ $\Big(\Big[\mathbf{CPatt\ \ C_1\ us_r@\ p_{r,1}},\ldots,p_{r,n}\Big],t_r\Big),$ $\Big(\Big[\mathbf{VPatt\ \ v}_{\text{r+1}}\mathbf{@\ p}_{\text{r+1},\mathbf{1}},\ldots,p_{\text{r+1},n}\Big],t_{\text{r+1}}\Big),$ $\vdots$ $\Big]\ \Big]\ \ \ default\ \ term$ **The above compile function can be rewritten to:** $compile\ \ [u_1,\ldots,u_n]$ $\Bigg[$ $\Bigg[\ \Big(\Big[\mathbf{VPatt\ \ v_1@\ p_{1,1}},\ldots,p_{1,n}\Big],t_1\Big),$ $\vdots$ $\Big(\Big[\mathbf{VPatt\ \ v_q@\ p_{q,1}},\ldots,p_{q,n}\Big],t_q\Big)\ \Bigg],$ $\Bigg[\Big(\Big[\mathbf{CPatt\ \ C_1\ us}_{\text{q+1}}\mathbf{@\ p}_{\text{q+1},\mathbf{1}},\ldots,p_{\text{q+1},n}\Big],t_{\text{q+1}}\Big),$ $\vdots$ $\Big(\Big[\mathbf{CPatt\ \ C_1\ us_r@\ p_{r,1}},\ldots,p_{r,n}\Big],t_r\Big)\ \Bigg],$ $\vdots$ $\Bigg]\ \ default\ \ term$	$compile\ \Big([u_1,\ldots,u_n]\Big)$ $\Big[\ \Big[$ $\Big(\Big[\mathbf{VPatt\ \ v_1@\ p_{1,1}},\ldots,p_{1,n}\Big],t_1\Big),$ $\vdots$ $\Big(\Big[\mathbf{VPatt\ \ v_q@\ p_{q,1}},\ldots,p_{q,n}\Big],t_q\Big)$ $\Big]\ \Big]$ $\Bigg($ $compile\ \Big([u_1,\ldots,u_n]\Big)$ $\Big[\ \Big[$ $\Big(\Big[\mathbf{CPatt\ \ C_1\ us}_{\text{q+1}}\mathbf{@\ p}_{\text{q+1},\mathbf{1}},\ldots,p_{\text{q+1},n}\Big],t_{\text{q+1}}\Big),$ $\vdots$ $\Big(\Big[\mathbf{CPatt\ \ C_1\ us_r@\ p_{r,1}},\ldots,p_{r,n}\Big],t_r\Big)$ $\Big]\ \Big]$ $\Bigg($ $\vdots$ $compile\ \Big([u_1,\ldots,u_n]\Big)$ $\Big[\ \Big[\ldots\Big]\ \Big]$ $\vdots$ $\Bigg)$ $\Bigg)$

Table 7: Mixture Rule

Before	**After**
$match\ \ [\,]$ $\Big[$ $([\,],\ \ t_1),$ $\vdots$ $([\,],\ \ t_n)$ $\Big]$ $default\ \ term$	t_1
$match\ \ [\,]\ \ [\,]\ \ default\ \ term$	$default\ \ term$

Table 8: Empty Rules