Compilation of Pattern Matching

June 26, 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 to type infer 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 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 gets rid of 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. The *pattern-matching compiler algorithm* was first given by Lennart Augustsson. However, the version of the algorithm described in the thesis and used in MPL's implementation is due to Philip Wadler.

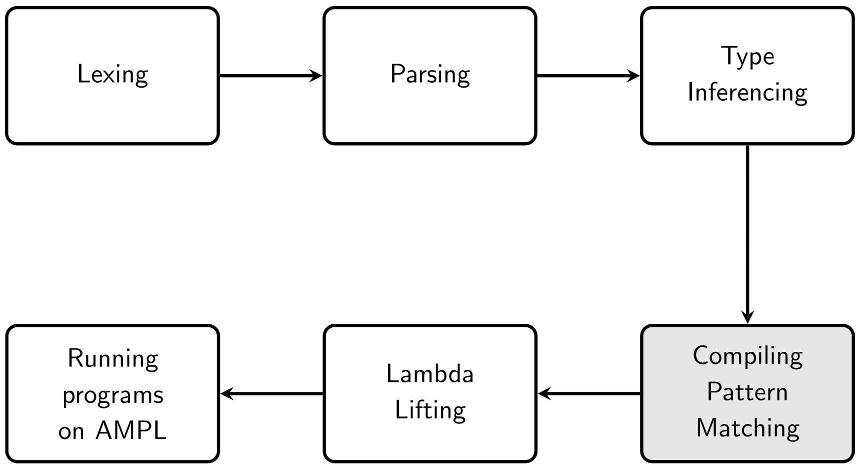

Figure 1: Compilation Stages : Pattern Matching Compilation

2 Pattern Matching Compiler Algorithm

MPL allows the programmer to define readable functions with the *pattern-matching* syntax. The general scheme for defining functions with pattern-matching has been described in Figure 2. Function f_1 has been defined with m pattern-matching lines. Each pattern-matching line has n patterns on the left side of the arrow and a sequential term on the right hand side of the arrow. A pattern can either be a variable pattern or constructor pattern. As the name suggests, a variable pattern uses a variable as pattern and a constructor pattern uses the constructors of a data type as a pattern. Pattern $p_{i,j}$ is the j^{th} pattern on the i^{th} pattern-matching line. t_i is the corresponding sequential term for the pattern-matching line i.

For a function defined with pattern-matching to be well formed, the following conditions need to hold:

- The number of patterns in all the pattern-matching lines need to be equal.
- In any given column of all the pattern-matching lines, one can have variable patterns, constructor patterns or the mixture of the two. However, all the constructors of the constructor patterns must belong to the same data type.

$$
\begin{array}{l}
fun \;\; f_1 = \\
\quad \begin{array}{llll}
p_{1,1}, & \dots \;, & p_{1,n} & \rightarrow \;\; t_1 \\
p_{2,1}, & \dots \;, & p_{2,n} & \rightarrow \;\; t_2 \\
\vdots & \;\;\vdots & & \;\;\;\vdots \\
p_{m,1}, & \dots \;, & p_{m,n} & \rightarrow \;\; t_m
\end{array}
\end{array}
$$

Figure 2: Function Defintion with Pattern Matching

The *pattern-matching compiler algorithm* takes a function definition that uses *pattern-matching* syntax and compiles it to a definition that uses case construct instead. This has been illustrated through the example of *append* function in Table 1.

In the next few sections, the technical details of the algorithm have been provided.

2.1 Overview of Pattern Matching Compiler Algorithm

The pattern matching compilation algorithm is described with the help of **match** notation. **match** helps in a clean presentation 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 **match** notation. Section 3 describes the **match** notation in details.

Function defintion with pattern-matching	**Function definition with case construct**
`fun append =`	`fun append =`
`  [],   ys   -> ys`	`  xl,yl ->`
`  x:xs, ys   -> x:append (xs,ys)`	`    case x of`
	`      Nil -> yl`
	`      Cons(x,xs) -> Cons(x,append(xs,yl))`

Table 1: Example of Pattern-Matching Compilation

- Define reduction rules for the **match** notation. A reduction rule is a syntactic rule to transform the match notation.
- Apply the reduction rules recursively till a normal form is achieved. The reduction rules and the normal form have been described in Section 5.

The normal form of **match** will the produce a new function body. The new function body:

- uses `case` instead of pattern-matching syntax that the input function body used.
- is equivalent to the input function body.

Using the algorithm sketch described here, the pattern-matching compiled version of function f_1, will be:

$$\begin{aligned}&fun \;\; f_1 =\\ &\quad v_1, v_2, \ldots, v_n = \textbf{normal form } (f_1 \; \textit{body})\end{aligned}$$

Here $v_1, v_2, \ldots, v_n$ are n fresh variables. Number of free variables is equal to the number of patterns in a pattern matching row.

3 match notation

Body of function f_1 (defined in Figure 2) represented with the **match** notation has been shown in Figure 3. As can be seen in Figure 3, **match** is a function that takes three arguments:

- First argument is a list of fresh variables. The number of fresh variables is the same as the number of patterns in any given pattern-matching line. In the given representation $[u_1, u_2, \ldots, u_n]$ is the list of n fresh variables because the function body of f_1 has n patterns in every pattern-matching line.
- Second argument is the list of pair of patterns and terms corresponding to every patetrn-matching line in the function body.
- Third argument is a default sequential term to be used with the missing constructors of a data type in pattern-matching.

$$
\begin{array}{l}
match\ \ [u_1, u_2, \ldots, u_n] \\
\quad [\\
\quad\quad ([\, p_{1,1},\ \ldots\ ,\ p_{1,n}\,],\ \ t_1) \\
\quad\quad ([\, p_{2,1},\ \ldots\ ,\ p_{2,n}\,],\ \ t_2) \\
\quad\quad\quad\quad \vdots \\
\quad\quad ([\, p_{m,1},\ \ldots\ ,\ p_{m,n}\,],\ t_m) \\
\quad] \\
\quad E
\end{array}
$$

Figure 3: Function Defintion with Pattern Matching

4 Configurations of match

There are four reduction rules for **match** corresponding to four different configurations that can occur inside it. These configurations are based on the second argument of **match** which contains the list of patterns for the different pattern matching lines along with the corresponding sequential term.

The first three configurations differentiate the type of patterns in the first column in every pattern-matching line, **variable, constructor or mixed**. The fourth case corresponds to the configuration where the pattern list corresponding to every pattern-matching line is empty or the second argument of **match** function is empty.

4.1 Variable First Patterns

In this configuration, all the patterns in the first column of pattern-matching lines are **variable patterns**. In Figure 4, the first patterns v and w are both variable patterns.

$$
\begin{array}{l}
match\ \ [u_1, u_2] \\
\quad [\\
\quad\quad ([\, \mathbf{v},\ Nil\,],\ \ v), \\
\quad\quad ([\, \mathbf{w},\ Cons\ (x, xs)\,],\ \ w * w) \\
\quad] \\
\quad E
\end{array}
$$

Figure 4: Example : Variable First Patterns

4.2 Constructor First Patterns

In this configuration, all the patterns in the first column of the pattern-matching lines are **constructor patterns**. In Figure 5, the first patterns Nil and $Cons$ are both constructor patterns.

$$\begin{array}{ll} match & [u_1, u_2] \\ & [\\ & \quad (\ [\ \mathbf{Nil},\ v\ ],\ \ v), \\ & \quad (\ [\ \mathbf{Cons}\ (\mathbf{x}, \mathbf{xs}),\ w\ ],\ \ w * w) \\ &] \\ & E \end{array}$$

Figure 5: Example : Constructor First Patterns

4.3 Mixed First Patterns

In this configuration, the patterns obtained from the first column of the pattern-matching lines are a **mixture of variable and constructor patterns**. In Figure 6, the first patterns are Nil, a constructor pattern and x, a variable pattern.

$$\begin{array}{ll} match & [u_1, u_2] \\ & [\\ & \quad (\ [\ \mathbf{Nil},\ v\ ],\ \ v) \\ & \quad (\ [\ \mathbf{x},\ w\ ],\ \ w * w) \\ &] \\ & E \end{array}$$

Figure 6: Example : Mixed First Patterns

4.4 Empty Configuration

Corresponding to this configuration, the normal form of **match** is obtained. This configuration takes two forms:

- Pattern list in all the elements of the second argument of **match** are empty. This configuration is shown in Figure 7.

$$\begin{array}{ll} match & [\,] \\ & [\\ & \quad (\ [\,],\ \ v) \\ & \quad (\ [\,],\ \ w * w) \\ &] \\ & E \end{array}$$

Figure 7: Example : Empty Configuration (Pattern Lists are Empty)

- The second argument of **match** is an empty list. This configuration is show in Figure 8.

$$match \;\; [\,] \;\; [\,] \;\; E$$

Figure 8: Empty Configuration : Second Argument is Empty

5 Reduction Rules for match

Reduction rules corresponding to the four **match** configurations have been listed below:

5.1 Variable Rule

This is the reduction rule corresponding to the **Variable First Patterns** configuration. The reduction rule for this case is provided in Table 2. In this reduction rule, $t_i[\ u/p_i\ ]$ signifies that all the instances of variable p_i inside t_i have been replaced by u.

Before	**After**
$match \;\; [u : us]$	$match \;\; us$
$[$	$[$
$(\ [\ p_1,\ \dots\ ,\ ps_1\ ],\ \ t_1)$	$(\ ps_1\ ,\ t_1[\ u/p_1\ ]\ )$
$(\ [\ p_2,\ \dots\ ,\ ps_2\ ],\ \ t_2)$	$(\ ps_2\ ,\ t_2[\ u/p_2\ ]\ )$
$\vdots$	$\vdots$
$(\ [\ p_{m,1},\ \dots\ ,\ p_{m,n}\ ],\ t_m)$	$(\ ps_m\ ,\ t_m[\ u/p_m\ ]\ )$
$]$	$]$
E	E

Table 2: Variable Rule

Applying this reduction rule to the **Variable First Pattern** example from Figure 4 results in the transformation as shown in Table 3.

5.2 Constructor Rule

This is the reduction corresponding to the **Constructor First Patterns** configuration. This rule is more complicated than the **Variable rule** and requires that the second argument of **match**, which is a list of pair of patterns and their corresponding sequential term, be represented in a particular format before the **Constructor Rule** can be applied. Section 5.2.1 deals with the reformatting of the **match**.

5.2.1 Reformatting match for Constructor Rule

Suppose the constructor patterns in the first column of the different pattern-lines are constructors of a data type d. Suppose d has m constructors, say $C_1, \dots, C_m$. The list of pairs of patterns and term can then be partitioned into m parts such that every part consist of the pairs whose first

Before	**After**
$match\ \ [u_1, u_2]$ $[$ $(\ [\ \mathbf{v},\ Nil\ ],\ \ v),$ $(\ [\ \mathbf{w},\ Cons\ (x, xs)\ ],\ \ w * w)$ $]$ E	$match\ \ [u_2]$ $[$ $(\ [\ Nil\ ],\ \ u_1),$ $(\ [\ Cons\ (x, xs)\ ],\ \ u_1 * u_1)$ $]$ E

Table 3: Variable Reduction Rule Example

$$match\ \ (u : us)\ \ (qs_1 \mathbin{++} \ldots \mathbin{++} qs_m)\ \ E$$

Figure 9: match Representation : Partitioning Constructors

pattern start with the same constructor. Let these partitions be $qs_1, qs_2, \ldots, qs_m$. The **match** can be represented in a format as shown in Figure 9.

Here, ++ is the list *append* function. qs_i consists of all the pairs correponding to a pattern-matching line which have constructor C_i as the first pattern. Every qs_i is of the form shown in Figure 10.

The internal representation of qs_i in Figure 10 conveys the following:

- There are j pattern-matching lines starting with C_i.
- ps' in the representation $C\ ps'$ stands for the list of the arguments that the constructor C_i takes.
- ps in the representation $((C\ ps') : ps)$ represents the remaining pattern list of a pattern-

$$\left[\begin{array}{c} \left(\left((C_i\ \ ps'_{i,1}) : ps_{i,1}\right), t_{i,1}\right), \\ \vdots \\ \left(\left((C_i\ \ ps'_{i,j}) : ps_{i,j}\right), t_{i,j}\right) \end{array}\right]$$

Figure 10: Internal Representation of qs_i

matching line except the last one.

Function Definition	**match Representation for *someFun* function body**

`fun someFun =`
`   []   ,[]   -> []`
`   []   ,ys   -> ys`
`   x:xs ,[]   -> xs`
`   x:xs ,y:ys -> ys`
	$match\ \ [u_1, u_2]$
	$[$
	$(\ [\ Nil,\ Nil\ ],\ Nil),$
	$(\ [\ Nil,\ ys\ ],\ ys),$
	$(\ [\ Cons\ (x, xs),\ Nil\ ],\ ys),$
	$(\ [\ Cons\ (x, xs),\ Cons\ (y, ys)\ ],\ ys)$
	$]$
	E

Reformatted match for *someFun*	
$match\ \ [u_1, u_2]$ $\Bigg($ $\Big[\ (\ [Nil,\ Nil],\ Nil),\ (\ [Nil,\ ys\ ],\ ys)\ \Big]\ ++$ $\Big[\ (\ [Cons\ (x, xs),\ Nil],\ Nil),\ (\ [Cons\ (x, xs), Cons\ (y, xs)],\ ys\ )\Big]$ $\Bigg)$ E	

Table 4: Example : Reformatting match for *someFun*

For example - In Table 4, the **match** representation for function $someFun$ has been reformatted based on the format described in Figure 9.

Nil and $Cons$ are two constructors corresponding to the $List$ data type. There are two pattern matching lines with Nil as the first constructor and two with $Cons$ as the first constructor.

If the reformatted **match** representation of $somefun$ is now compared to the **match** format described in Figure 9, then :

$$[\ (\ [Nil,\ Nil],\ Nil),\ (\ [Nil,\ ys\ ],\ ys)\ ]\ \ is\ \ \mathbf{qs_1}$$
$$[\ (\ [Cons\ (x, xs),\ Nil],\ Nil),\ (\ [Cons\ (x, xs), Cons\ (y, xs)],\ ys\ )]\ \ is\ \ \mathbf{qs_2}$$

5.2.2 Applying Constructor Rule to the Reformatted match

Once the match has been reformatted, the **Constructor Rule** can now be described as shown in Table 5. Here, each qs_i is of the form described in Figure 10, qs'_i is of the form described in Figure

$$\begin{bmatrix} \\ ((ps'_{i,1} \ {+}{+} \ ps_{i,1}), t_{i,1}) \\ \vdots \\ ((ps'_{i,1} \ {+}{+} \ ps_{i,1}), t_{i,1}) \\ \end{bmatrix}$$

Figure 11: Internal Representation of qs'_i

11 and $us_{i'}$ is a list of fresh variables. The number of fresh variables in the list $us_{i'}$ is the same as the number of arguments that the constructor C_i takes as input.

Before	**After**
$match \ \ (u : us) \ \ (qs_1 \ {+}{+} \ldots {+}{+} \ qs_m) \ \ E$	$case \ \ u \ \ of$ $C_1 \ \ us'_1 \ \ \to \ \ match \ \ (us'_1 \ {+}{+} \ us) \ \ qs'_1 \ \ E$ $\vdots$ $C_1 \ \ us'_m \ \ \to \ \ match \ \ (us'_m \ {+}{+} \ us) \ \ qs'_m \ \ E$

Table 5: Constructor Rule

Table 6 demonstrates an example of reduction using the **Constructor Rule**. The **match** representation in the before column of the table is the reformatted **match** notation of the function *someFun* taken from Table 4.

5.3 Mixture Rule

This is the reduction rule is corresponding to the **Mixed First Patterns** configuration.

5.3.1 Reformatting match for Mixture rule

Suppose the **match** representation to be reduced is of the form

$$match \ \ us \ \ qs \ \ E$$

The pair list qs can be partitioned into m lists such that it is of the form show in Figure 12. Each qs_i should start either with a **Variable Pattern** or a **Constructor Pattern**.

Function Definition	**match representation of the function body**
$match\ [u_1, u_2]\ \Big(\big[([Nil,\ Nil\ ],\ Nil),\ ([Nil,\ ys\ ],\ ys) \big] \mathbin{++} \big[([Cons\ (x,xs),\ Nil],\ Nil),\ ([Cons\ (x,xs), Cons\ (y,xs)],\ ys\ ) \big] \Big)\ E$	$\textbf{case}\ u_1\ of$ $\quad \textbf{Nil}\ \rightarrow$ $\quad\quad match\ [u_2]\ \big[([\ Nil\ ],\ Nil),\ ([\ ys\ ],\ ys) \big]\ E$ $\quad \textbf{Cons}\ (\mathbf{u_3}, \mathbf{u_4})\ \rightarrow$ $\quad\quad match\ [u_3, u_4, u_1]\ \big[([x, xs,\ Nil],\ Nil),\ ([x, xs, Cons\ (y,xs)],\ ys) \big]\ E$

Table 6: Example : Constructor Rule

$$qs \;=\; qs_1 \;\mathbin{++}\; qs_2 \;\mathbin{++}\; \ldots \;\mathbin{++}\; qs_m$$

Figure 12: Mixture Rule : Reformatting Scheme for match in Mixture Rule

Before	**After**
$match\ \ us\ \ (qs_1 \mathbin{++} \ldots \mathbin{++} qs_m)\ \ E$	$match\ \ us\ \ qs_1\ \ (match\ \ us\ \ qs_2\ \ (\ \ldots\ \ (match\ \ us\ \ qs_m)\ \ldots\ ))$

Table 7: Mixture Rule

5.3.2 Applying Mixture Rule to the Reformatted match

Once the **match** has been reformatted based on the scheme for **Mixture Rule** described in Section 5.3.1, the reduction rule corresponding to **Mixed First Patterns** can now be described. This has been done in Table 7.

Applying the **Mixture Rule** to the example in Figure 6, one starts with reformatting the **match** to the scheme suggested in Figure 12. As a result of the reformatting, the **match** representation shown in Figure 13 is obtained. Now the **Mixture Rule** can be applied to this reformatted match as shown in Table 8.

$$
\begin{array}{l}
match\ \ [u_1, u_2] \\
\quad \Bigg(\\
\qquad \Big[\ (\ [Nil,\ v],\ v)\ \Big]\ +\!\!+ \\
\qquad \Big[\ (\ [x,\ w],\ w * w)\ \Big] \\
\quad \Bigg) \\
\quad E
\end{array}
$$

Figure 13: Mixture Rule : Reformatted match

Function Definition	**match representation of the function body**
$match\ \ [u_1, u_2]\ \Bigg(\ \Big[\ (\ [Nil,\ v],\ v)\ \Big]\ +\!\!+\ \Big[\ (\ [x,\ w],\ w * w)\ \Big]\ \Bigg)\ E$	$match\ \ [u_1, u_2]\ \Big[\ (\ [Nil,\ v],\ v)\ \Big]\ \Bigg(\ match\ \ [u_1, u_2]\ \Big[\ (\ [x,\ w],\ w * w)\ \Big]\ \ E\ \Bigg)$

Table 8: Example : Constructor Rule

5.4 Empty Rules and Normal Form of match

Empty Rules are applied when an **Empty configuration** is obtained. **Empty Configuration** signals that there are there are no more patterns to be compiled and thus marks the end of reduction process with the generation of the normal form for the **match** function. The **Empty Rules** have been described in Table 9.

6 Example

append function was defined at the beginning of this chapter with pattern-matching and with case construct (Table 1) as the motivating example for *pattern-matching compilation.* In this section, the tools described in the previous sections of the chapter, i.e **match** function, its reduction rules and normal form, are used to show the step by step translation of *append* function body with pattern matching to the one with case construct.

Before	**After**
$match\ \ [\,]\ \left[\begin{array}{l} ([\,],\ \ E_1), \\ \vdots \\ ([\,],\ \ E_n) \end{array} \right]\ E$	E_1
$match\ \ [\,]\ \ [\,]\ \ E$	E

Table 9: Empty Rules

Table 10 describes the overview of the strategy used in the compilation of pattern-matching of *append*. The function body of *append* is converted to its **match** representation. The normal form of the **match** representaion is then found and plugged in the skeleton provided in step 2 of Table 10.

The calculation of the normal form for the **match** representation of the body of *append* has been described step wise in Table 11. To keep the table readable, two **match** functions generated in step 3 of Table 11 have been normalised in their own tables. **match** corresponding to the **Nil** constructor has been normalised in Table 12 and the **match** corresponding to the **Cons** constructor has been normalised in Table 13. These normal forms are substituted directly in step 4 and step 5 respectively of the Table 11 for their corresponding **match** functions.

Once the normal form of the **match** function corresponding to the *append* function body is found, the function definition with case is be easily obtained as can be seen in Figure 14.

```
fun append =
  u1,u2 -> case u1 of
              Nil -> u3
              Cons (u3,u4) -> Cons (u3,append(u4,u2))
```

Figure 14: *append* definition with case

***append* definition with pattern-matching**

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

Step 1 : match Representation of the body of *append*

$$
\begin{array}{l}
match\ \ [u_1, u_2] \\
\quad \Bigg[\\
\quad \Big([Nil,\ ys\ ],\ ys\Big), \\
\quad \Big([Cons(x, xs), ys],\ Cons\big(x, append\ (xs, ys)\big)\Big) \\
\quad \Bigg] \\
\quad E
\end{array}
$$

Step 2 : *append* definition with case

```
fun append =
  u1,u2 -> normal form (match Rep. of body)
```

Table 10: Overview of Pattern-Matching Compilation of *append*

1 : Reformatting match for Constr. Rule	**2 : Applying Constr. Reduction Rule**
$match\ \ [u_1,u_2]$ $\Big[$ $\big([Nil,\ ys\ ],\ ys\big),$ $\big([Cons(x,xs),ys],\ Cons\big(x,append\ (xs,ys)\big)\big)$ $\Big]$ E	$match\ \ [u_1,u_2]$ $\Big($ $\Big[\big(\ [Nil,\ ys\ ],\ ys\big)\Big]$ ++ $\Big[\big(\ [Cons(x,xs),ys],\ Cons(x,append(xs,ys)\big)\Big]$ $\Big)$ E
2 : Reducing *Nil* match (see Table 12)	**5 : Reducing *Cons* match (see Table 13)**
$case\ \ u_1\ \ of$ $Nil\ \ \rightarrow$ $match\ \ [u_2]\ \Big[\ \big([\ ys\ ],ys\big)\ \Big]\ \ E$ $Cons\ (u_3,u_4)\ \ \rightarrow$ $match\ \ [u_3,u_4,u_2]$ $\Big[$ $\Big([\ x,xs,ys\ ],\ Cons\ \big(x,\ append\ (xs,ys)\ \big)\Big)$ $\Big]\ \ E$	$case\ \ u_1\ \ of$ $Nil\ \ \rightarrow\ \ u_2$ $Cons\ (u_3,u_4)\ \ \rightarrow$ $match\ \ [u_3,u_4,u_2]$ $\Big[$ $\Big([\ x,xs,ys\ ],\ Cons\ \big(x,\ append\ (xs,ys)\ \big)\Big)$ $\Big]\ \ E$
$case\ \ u_1\ \ of$ $Nil\ \ \rightarrow\ \ u_2$ $Cons\ (u_3,u_4)\ \ \rightarrow\ \ Cons\ \big(u_3,\ append\ (u_4,u_2)\ \big)$	

Table 11: 5 : *append* body with case

1 : Variable Rule	**2 : Empty Rule**	**3 : Normal Form**
$match\ \ [u_2]\ \Big[\ \big([\ ys\ ],ys\big)\ \Big]\ \ E$	$match\ \ [\ ]\ \Big[\ \big([\ ],u_2\big)\ \Big]\ \ E$	u_2

Table 12: Reduction of **match** (corresponding to Nil, step 3, Table 11)

$\mathbf{1:}$ **Variable Rule**	$\mathbf{2:}$ **Variable Rule**
$match\ \ [u_3, u_4, u_2]$ $\Big[$ $\Big([\ x, xs, ys\ ],\ Cons\ \big(x,\ append\ (xs, ys)\ \big)\Big)$ $\Big]\ E$	$match\ \ [u_4, u_2]$ $\Big[$ $\Big([\ xs, ys\ ],\ Cons\ \big(u_3,\ append\ (xs, ys)\ \big)\Big)$ $\Big]\ E$
Variable Rule	**Empty Rule**
$match\ \ [u_2]$ $\Big[$ $\Big([\ ys\ ],\ Cons\ \big(u_3,\ append\ (u_4, ys)\ \big)\Big)$ $\Big]\ E$	$match\ \ [\ ]$ $\Big[$ $\Big([\ ],\ Cons\ \big(u_3,\ append\ (u_4, u_2)\ \big)\Big)$ $\Big]\ E$
$\mathbf{5:}$ **Normal Form**	
$Cons\ \big(u_3,\ append\ (u_4, u_2)\ \big)$	

Table 13: Reduction of **match** (corresponding to Cons, step 3, Table 11)