Compilation of Pattern-Matching

July 13, 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.

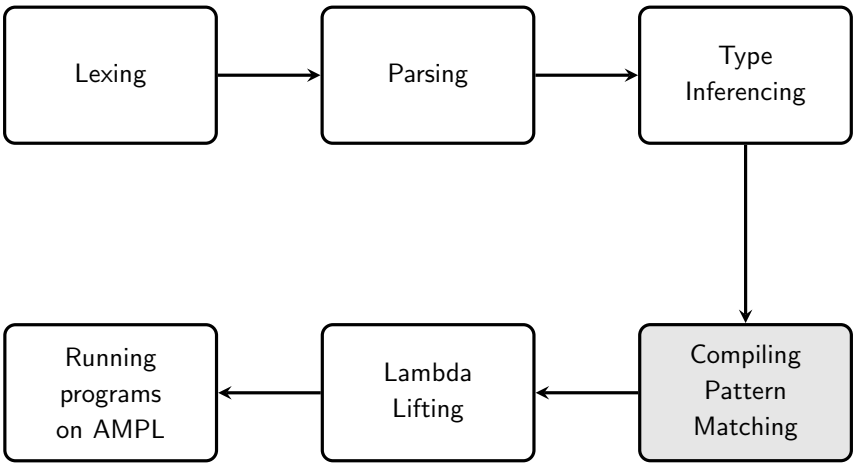

Figure 1: Compilation Stages : Pattern Matching Compilation

2 Examples of 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 pairs of **n** patterns and a sequential term. A line **i** consisting of the pair of patterns $\mathbf{p_{i,1}}, \ldots, \mathbf{p_{i,n}}$ and the term $\mathbf{t_i}$ is called a **pattern-matching line**.

$$
\begin{aligned}
&fun \;\; f = \\
&\quad p_{1,1}, \;\; \ldots \;\; , \; p_{1,n} \;\; \rightarrow \;\; t_1 \\
&\qquad \ldots \\
&\quad p_{m,1}, \;\; \ldots \;\; , \; p_{m,n} \;\; \rightarrow \;\; t_m
\end{aligned}
$$

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

A function can have a mixture of these patterns in any given pattern matching line. The constructor, record and product patterns may have other patterns in their body.

The pattern-matching compilation algorithm gets rid of all the constructor patterns from a function definition that uses pattern-matching syntactic sugar and replaces it with `case` statements.

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).

pairEx 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 defintion with pattern-matching	Function definition with case construct

`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 pairEx::<[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 pairEx::<[Int],[Int]>,<Int,Int> ->`
	`  (Int,Int) =`
	`  <xls,yls>,<p,q> ->`
	`  case xls of`
	`  Nil ->`
	`  case yls of`
	`  Nil  -> <p,q>`
	`  Cons(y,ys) -> <p,y>`
	`  Cons(x,xs) ->`
	`  case yls of`
	`  Nil  -> <x,q>`
	`  Cons(y,ys) -> <x,y>`

`fun recEx::InfList([A]) ->`
`  InfList([A]) =`
`  (: Head := [],Tail := t :) ->`
`  (: Head := [],Tail := recEx(t) :)`
 	
`  (: Head := a:as,Tail := t :) ->`
`  (: Head := as,Tail := recEx(as) :)`
	`fun recEx::InfList([A]) ->`
	`  InfList([A]) =`
	`  (: Head := x,Tail := t :) ->`
	`  case x of`
	`  [] ->`
	`  (: Head := [],Tail := recEx(t) :)`
	`  x:xs ->`
	`  (: Head := xs,Tail := recEx(xs) :)`

Table 1: Example of Pattern-Matching Compilation

3 Pattern Matching Compiler Algorithm

In this section, the algorithm for step wise compilation of patterns in a function is described.

3.1 Overview of Pattern Matching Compiler Algorithm

The pattern matching compilation algorithm is described with the help of **match** notation. **match** notation 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 4 describes the **match** notation in details. MPL allows for product patterns and record patterns in addition to variable and constructor patterns. The translation of various patterns to **match** has been described in Section 5.

- 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 rule that is applied to a **match** notation dependes on its configuration. The different configurations of **match** have been described in Section 6. The reduction rules and the normal form have been described in Section 7.

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.

4 match notation

The structure of **match** notation has been shown in Figure 3.

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

Figure 3: Function Defintion with Pattern Matching

As can be seen in Figure 3, **match** is a function that takes three arguments:

- First argument is a list of fresh variables.
- Second argument is the list of pair of patterns and a term.
- Third argument is a sequential term. Usually it is the default term to be used with the missing constructors of a data type in pattern-matching.

5 Converting Function Body to match Function

This is a step in pattern-matching compilation where the function body of a function definition consisting of several pattern-matching lines is converted into **match** function which can then be reduced. This section describes the conversion of different types of patterns to their corresponding **match** notation.

5.1 Converting Function Body With Constructor or Variable Patterns to match Function

Function definitions that have just variable or constructor patterns are easy to translate to **match**. The pattern-matching lines are used directly as the second argument of the match function. Table 2 shows this conversion of variable or constructor patterns to **match** and also outlines the structure of the pattern-matching compiled function.

Function with pattern-matching	**Function with pattern-matching compiled**
$fun \;\; f1 =$ $\quad p_{1,1}, \ldots, p_{1,n} \;\; \rightarrow \;\; t_1$ $\quad\quad\quad \vdots$ $\quad p_{m,1}, \ldots, p_{m,n} \;\; \rightarrow \;\; t_m$	$fun \;\; f1 =$ $\quad u_1, \ldots, u_n \;\; \rightarrow$ $\quad\quad$ **normal Form of** $\quad\quad\quad match \;\; [u_1, \ldots, u_n]$ $\quad\quad\quad [$ $\quad\quad\quad\quad (\; [\; p_{1,1}, \ldots, p_{1,m} \;], \;\; t_1)$ $\quad\quad\quad\quad\quad\quad \vdots$ $\quad\quad\quad\quad (\; [\; p_{m,1}, \;\; \ldots \;\; , \;\; p_{m,n} \;], \;\; t_m)$ $\quad\quad\quad]$ $\quad\quad\quad E$

Table 2: Translation to match: Variable/Constructor Patterns

5.2 Converting Function Body with Product Patterns to match Function

The product patterns in MPL are represented to their corresponding **match** function using the schemes shown in Table 3. The first row of Table 3 shows the case where all the patterns in the pattern matching lines have been expressed as product pattern syntax. In this case, the match takes **n** fresh variables, which is the same number as the number of elements in the product pattern. The list of pair of patterns inside the product pattern along with the corresponding sequential term forms the second argument of the match.

Second row of Table 3 represents a case where the pattern in the last pattern matching line is not a product pattern but a variable pattern. In this case **p** is substituted with a product of **n** fresh variables (where the product being pattern matched on is a **n** tuple) both in the pattern and in the sequential term of the last pattern matching line. The notation $< \mathbf{u_1}, \ldots, \mathbf{u_n} > / \mathbf{p}$ in the pattern-mathcing compiled column of the second row means that product $< \mathbf{u_1}, \ldots, \mathbf{u_n} >$ has been replaced for **p** inside the sequential term $\mathbf{t_m}$.

5.3 Converting Function Body with Record Patterns to match Function

Table 4 shows the representation of record patterns to their corresponding match function. This case is almost identical to that of product patterns. In the table, there are **n** destructors $\mathbf{D_1}, \ldots, \mathbf{D_n}$ each having a corresponding pattern. **n** fresh variables are generated and used as variable patterns

Function with pattern-matching	**Function with pattern-matching compiled**
$fun\ \ f1 =$ $\quad < p_{1,1}, \ldots, p_{1,n} > \quad \rightarrow \quad t_1$ $\qquad\qquad \vdots$ $\quad < p_{m,1}, \ldots, p_{m,n} > \quad \rightarrow \quad t_m$	$fun\ \ f1 =$ $\quad < u_1, \ldots, u_n > \quad \rightarrow$ $\quad\quad$ **normal Form of** $\quad\quad\quad match\ \ [u_1, \ldots, u_n]$ $\quad\quad\quad [$ $\quad\quad\quad\quad ([\ p_{1,1}, \ldots, p_{1,m}\ ],\ \ t_1)$ $\quad\quad\quad\quad\quad\quad \vdots$ $\quad\quad\quad\quad ([\ p_{m,1},\ \ \ldots\ ,\ p_{m,n}\ ],\ t_m)$ $\quad\quad\quad]$ $\quad\quad\quad E$
$fun\ \ f1 =$ $\quad < p_{1,1}, \ldots, p_{1,n} > \quad \rightarrow \quad t_1$ $\qquad\qquad \vdots$ $\qquad p \qquad\qquad\quad \rightarrow \quad t_m$	$fun\ \ f1 =$ $\quad < u_1, \ldots, u_n > \quad \rightarrow$ $\quad\quad$ **normal Form of** $\quad\quad\quad match\ \ [u_1, \ldots, u_n]$ $\quad\quad\quad [$ $\quad\quad\quad\quad ([\ p_{1,1}, \ldots, p_{1,m}\ ],\ \ t_1)$ $\quad\quad\quad\quad\quad\quad \vdots$ $\quad\quad\quad\quad ([\ u_1,\ \ldots\ ,\ u_n\ ],\ \ t_m\ [< u_1, \ldots, u_n > /p])$ $\quad\quad\quad]$ $\quad\quad\quad E$

Table 3: Translation to match: Product Patterns

corresponding correspnding to the **n** destructors. The fresh variables $u_1, \ldots, u_n$ are also used as the first argument to the corresponding **match** function for the record pattern. The patterns assigned to the destructor and the sequential term for the pattern-matching line form the second argument of the **match** function.

6 Configurations of match Function

There are six 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 five configurations differentiate the type of patterns in the first column in every pattern-matching line, **variable, product, record, constructor or mixed**. The sixth configuration corresponds to the configuration where the pattern list corresponding to every pattern-matching

Function with pattern-matching	**Function with pattern-matching compiled**
$fun\ \ f1 =$ $(:\ D_1\ :=\ p_{1,1},\ \ldots,\ D_n\ :=\ p_{1,n}\ :)\ \ \rightarrow\ \ t_1$ $\vdots$ $(:\ D_1\ :=\ p_{m,1},\ \ldots,\ D_n\ :=\ p_{m,n}\ :)\ \ \rightarrow\ \ t_m$	$fun\ \ f1 =$ $(:\ D_1\ :=\ u_1,\ \ldots,\ D_n\ :=\ u_n\ :)\ \ \rightarrow$ **normal Form of** $match\ \ [u_1, \ldots, u_n]$ $[$ $(\ [\ p_{1,1}, \ldots, p_{1,m}\ ],\ \ t_1)$ $\vdots$ $(\ [\ p_{m,1},\ \ldots\ ,\ p_{m,n}\ ],\ t_m)$ $]$ E

Table 4: Translation to match: Record Patterns

line is empty or the second argument of **match** function is empty.

6.1 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.

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

Figure 4: Example : Variable First Patterns

6.2 Product First Patterns

In this configuration, all the the patterns in the first column of the pattern-matching lines are **product patterns**. In Figure 5, the first patterns $< \mathbf{Nil}, \mathbf{Nil} >$ and $< \mathbf{x}, \mathbf{y} >$ are both product patterns.

6.3 Record First Patterns

In this configuration, all the the patterns in the first column of the pattern-matching lines are **record patterns**. In Figure 6, the first patterns in the two pattern-matching lines are $(:\ \mathbf{Head}\ :=\ [\ ],\ \mathbf{Tail}\ :=\ \mathbf{t}\ :)$

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

Figure 5: Example : Product First Patterns

and (: **Head** := **a** : **as**, **Tail** := **t** :) . They are both record patterns of the **InfList** codata type which has been defined in Figure .

$$
\begin{array}{l}
match\ \ [u_1, u_2] \\
\quad [\\
\quad\ \ (\ [\ (:\ \mathbf{Head}\ :=\ [\ ],\ \mathbf{Tail}\ :=\ \mathbf{t}\ :),\ v\ ],\ \ v), \\
\quad\ \ (\ [\ (:\ \mathbf{Head}\ :=\ \mathbf{a} : \mathbf{as},\ \mathbf{Tail}\ :=\ \mathbf{t}\ :),\ w\ ],\ \ w * w) \\
\quad] \\
\quad E
\end{array}
$$

Figure 6: Example : Record First Patterns

6.4 Constructor First Patterns

In this configuration, all the patterns in the first column of the pattern-matching lines are **constructor patterns**. In Figure 7, the first patterns **Nil** and **Cons** are both constructor patterns.

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

Figure 7: Example : Constructor First Patterns

6.5 Mixed First Patterns

In this configuration, the patterns obtained from the first column of the pattern-matching lines are a **mixture of variable, product, record, or constructor patterns**. In Figure 8, the first

patterns are **Nil**, a constructor pattern and **x**, a variable pattern.

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

Figure 8: Example : Mixed First Patterns

6.6 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 9.

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

Figure 9: Example : Empty Configuration (Pattern Lists are Empty)

- The second argument of **match** is an empty list. This configuration is show in Figure 10.

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

Figure 10: Empty Configuration : Second Argument is Empty

7 Reduction Rules for match

Reduction rules corresponding to the six **match** configurations desacribed in Section 6 are listed below:

7.1 Variable Rule

This is the reduction rule corresponding to the **Variable First Patterns** configuration. The reduction rule for this case is provided in Table 5. In this after column of the reduction rule, $t_i[\ u/p_i\ ]$ signifies that all the instances of variable p_i inside term t_i have been replaced by u.

Before	**After**
$match\ \ [u_1, \ldots, u_n]$ $[$ $(\ [\ p_{1,1},\ \ldots\ ,\ p_{1,n}\ ],\ \ t_1)$ $\vdots$ $(\ [\ p_{m,1},\ \ldots\ ,\ p_{m,n}\ ],\ t_m)$ $]$ E	$match\ \ [u_2, \ldots, u_n]$ $[$ $(\ [\ p_{1,2},\ \ldots\ ,\ p_{1,n}\ ],\ t_1[\ u/p_1\ ]\ )$ $\vdots$ $(\ [\ p_{m,2},\ \ldots\ ,\ p_{m,n}\ ],\ t_m[\ u/p_m\ ]\ )$ $]$ E

Table 5: Variable Rule

Applying this reduction rule to the **Variable First Pattern** example from Figure 4 results in the transformation as shown in Table 6.

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 6: Variable Reduction Rule Example

7.2 Product Rule

This is the reduction rule corresponding to the **Product First Patterns** configuration. This rule has been shown in Table 7. In the table, the first product pattern in each pattern-matching line contains **a** patterns. The reduction rules takes the **a** patterns from the product in each pattern-matching line and appends it with the tail of the original pattern list on that pattern-matching line. **a** fresh variables,$\mathbf{u_{n+1}}, \ldots, \mathbf{u_{n+a+1}}$ are also generated and appended to the tail of the original fresh variable list.

Before	After
$match\ \ [u_1, \ldots, u_n]$ $\quad [$ $\quad\quad (\ [< p_{1,1,1}, \ldots, p_{1,1,a} >, p_{1,2}, \ldots, p_{1,n}\ ],\ t_1)$ $\quad\quad\quad\quad \vdots$ $\quad\quad (\ [< p_{m,1,1}, \ldots, p_{m,1,a} >, p_{m,2}, \ldots, p_{m,n}\ ],\ t_m)$ $\quad]$ $\quad E$	$match\ \ ([u_{n+1}, \ldots, u_{n+a+1}]$ ++ $[u_2, \ldots, u_n])$ $\quad [$ $\quad\quad (\ [p_{1,1,1}, \ldots, p_{1,1,a}, p_{1,2}, \ldots, p_{1,n}\ ],\ t_1)$ $\quad\quad\quad\quad \vdots$ $\quad\quad (\ [p_{m,1,1}, \ldots, p_{m,1,a}, p_{m,2}, \ldots, p_{m,n}\ ],\ t_m)$ $\quad]$ $\quad E$

Table 7: Product Rule

7.3 Record Rule

This is the reduction rule corresponding to the **Record First Patterns** configuration. This rule has been shown in Table 8. This rule is similar to the product rule which is not surprising because the products are stored in records internally.

Before	After
$match\ \ [u_1, \ldots, u_n]$ $\quad [$ $\quad\quad (\ [< D_1\ :=\ p_{1,1,1}, \ldots, D_a\ :=\ p_{1,1,a} >, p_{1,2}, \ldots, p_{1,n}\ ],\ t_1)$ $\quad\quad\quad\quad \vdots$ $\quad\quad (\ [< D_1\ :=\ p_{m,1,1}, \ldots, D_a\ :=\ p_{m,1,a} >, p_{m,2}, \ldots, p_{m,n}\ ],\ t_m)$ $\quad]$ $\quad E$	$match\ \ ([u_{n+1}, \ldots, u_{n+a+1}]$ ++ $[u_2, \ldots, u_n])$ $\quad [$ $\quad\quad (\ [p_{1,1,1}, \ldots, p_{1,1,a}, p_{1,2}, \ldots, p_{1,n}\ ],\ t_1)$ $\quad\quad\quad\quad \vdots$ $\quad\quad (\ [p_{m,1,1}, \ldots, p_{m,1,a}, p_{m,2}, \ldots, p_{m,n}\ ],\ t_m)$ $\quad]$ $\quad E$

Table 8: Product Rule

7.4 Constructor Rule

This is the reduction rule 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 7.4.1 deals with the reformatting of the **match**.

$$match \;\; (u : us) \;\; (qs_1 \; ++ \ldots ++ \; qs_m) \;\; E$$

Figure 11: match Representation : Partitioning Constructors

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

Figure 12: Internal Representation of qs_i

7.4.1 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 $C_1, \ldots, C_m$. 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 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 11.

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 12.

The internal representation of qs_i in Figure 12 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-matching line except the last one.

For example - In Table 9, the **match** representation for function *someFun* has been reformatted based on the format described in Figure 11.

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 de-

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

Table 9: Example : Reformatting match for *someFun*

scribed in Figure 11, then :

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

7.4.2 Applying Constructor Rule to the Reformatted match

Once the match has been reformatted, the **Constructor Rule** can now be described as shown in Table 10. Here, each qs_i is of the form described in Figure 12, qs_i' is of the form described in Figure 13 and $us_i\prime$ is a list of fresh variables. The number of fresh variables in the list $us_i\prime$ is the same as the number of arguments that the constructor C_i takes as input.

Table 11 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 9.

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

Table 10: Constructor Rule

$$\left[\begin{array}{l} \\ \quad \big((ps'_{i,1} \ \mathbin{++}\ ps_{i,1}), t_{i,1}\big) \\ \quad\quad \vdots \\ \quad \big((ps'_{i,1} \ \mathbin{++}\ ps_{i,1}), t_{i,1}\big) \\ \\ \end{array}\right]$$

Figure 13: Internal Representation of qs'_i

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

Table 11: Example : Constructor Rule

$$qs \;=\; qs_1 \;++\; qs_2 \;++\; \ldots \;++\; qs_m$$

Figure 14: Mixture Rule : Reformatting Scheme for match in Mixture Rule

7.5 Mixture Rule

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

7.5.1 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 14. Each qs_i should start either with a **Variable Pattern** or a **Constructor Pattern**.

Before	**After**
$match \;\; us \;\; (qs_1 \;++ \ldots ++\; qs_m) \;\; E$	$match \;\; us \;\; qs_1 \;\; (match \;\; us \;\; qs_2 \;\; (\;\ldots\; (match \;\; us \;\; qs_m) \;\ldots\;))$

Table 12: Mixture Rule

7.5.2 Applying Mixture Rule to the Reformatted match

Once the **match** has been reformatted based on the scheme for **Mixture Rule** described in Section 7.5.1, the reduction rule corresponding to **Mixed First Patterns** can now be described. This has been done in Table 12.

Applying the **Mixture Rule** to the example in Figure 8, one starts with reformatting the **match** to the scheme suggested in Figure 14. As a result of the reformatting, the **match** representation shown in Figure 15 is obtained. Now the **Mixture Rule** can be applied to this reformatted match as shown in Table 13.

7.6 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 14.

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

Figure 15: Mixture Rule : Reformatted match

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

Table 13: Example : Constructor Rule

8 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.

Table 15 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 15.

The calculation of the normal form for the **match** representation of the body of *append* has been described step wise in Table 16. To keep the table readable, two **match** functions generated in

Before	**After**
$match\ [\,]$ $\begin{bmatrix} ([\,],\ E_1), \\ \vdots \\ ([\,],\ E_n) \end{bmatrix}$ E	E_1
$match\ [\,]\ [\,]\ E$	E

Table 14: Empty Rules

step 3 of Table 16 have been normalised in their own tables. **match** corresponding to the **Nil** constructor has been normalised in Table 17 and the **match** corresponding to the **Cons** constructor has been normalised in Table 18. These normal forms are substituted directly in step 4 and step 5 respectively of the Table 16 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 16.

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

Figure 16: *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*
$match\ \ [u_1, u_2]$ $\Big[$ $\Big([Nil,\ ys\ ],\ ys\Big),$ $\Big([Cons(x, xs), ys],\ Cons\big(x, append\ (xs, ys)\big)\Big)$ $\Big]$ E
Step 2 : *append* definition with case
`fun append =` `  u1,u2 -> normal form (match Rep. of body)`

Table 15: 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 17)	**5 : Reducing *Cons* match (see Table 18)**
$case\ \ u_1\ \ of$ $Nil\ \ \rightarrow$ $\quad match\ \ [u_2]\ \ \Big[\ \big([\ ys\ ], ys\big)\ \Big]\ \ E$ $Cons\ (u_3, u_4)\ \ \rightarrow$ $\quad match\ \ [u_3, u_4, u_2]$ $\quad\quad \Big[$ $\quad\quad \Big([\ x, xs, ys\ ],\ Cons\ \big(x,\ append\ (xs, ys)\ \big)\Big)$ $\quad\quad \Big]\ \ E$	$case\ \ u_1\ \ of$ $Nil\ \ \rightarrow\ \ u_2$ $Cons\ (u_3, u_4)\ \ \rightarrow$ $\quad match\ \ [u_3, u_4, u_2]$ $\quad\quad \Big[$ $\quad\quad \Big([\ x, xs, ys\ ],\ Cons\ \big(x,\ append\ (xs, ys)\ \big)\Big)$ $\quad\quad \Big]\ \ E$
$case\ \ u_1\ \ of$ $Nil\ \ \rightarrow\ \ u_2$ $Cons\ (u_3, u_4)\ \ \rightarrow\ \ Cons\ \big(u_3,\ append\ (u_4, u_2)\ \big)$	

Table 16: 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 17: Reduction of **match** (corresponding to Nil, step 3, Table 16)

1 : Variable Rule	**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**
$match\ \ [u_2]$ $\Big[$ $\Big(\![\ ys\ ],\ Cons\ \big(u_3,\ append\ (u_4, ys)\ \big)\Big)$ $\Big]\ E$	$match\ \ [\,]$ $\Big[$ $\Big(\![\,],\ Cons\ \big(u_3,\ append\ (u_4, u_2)\ \big)\Big)$ $\Big]\ E$
5 : Normal Form	
$Cons\ \big(u_3,\ append\ (u_4, u_2)\ \big)$	

Table 18: Reduction of **match** (corresponding to Cons, step 3, Table 16)