Lambda Lifting Transformation

July 19, 2017

1 Sequential MPL

Sequential MPL consists of data type/codata type definitions and function definitions. Function definitions use pattern matching syntactic and use various sequential constructs in their body. Sequential MPL constructs can be divided into three parts: constructs that deal with data types , constructs that deal with codata types and constructs that deal with neither.

Data types and their constructs are described in Section 2. Codata types and their constructs are described in Section 3. Section 4 describes non data type/codata type constructs. Section 5 introduces pattern matching syntactic sugar in MPL and Section 6 describes how higher order functions can be obtained in MPL.

2 Data Type and Constructs for Data Type

MPL allows the programmers to define data types. The syntax of a data definition is:

$$\begin{aligned}\textbf{data}\;\; \mathbf{D}(\mathbf{A_1},\ldots,\mathbf{A_n})\; \rightarrow \mathbf{Z} \;\; = \;\; & \mathbf{C_1} : \mathbf{F_1}\;\tilde{\mathbf{P_1}}\;\mathbf{Z}\; \rightarrow \mathbf{Z} \\ & \vdots \\ & \mathbf{C_n} : \mathbf{F_n}\;\tilde{\mathbf{P_n}}\;\mathbf{Z}\; \rightarrow \mathbf{Z}\end{aligned}$$

Noticeable things about the data defintion are below:

- The name of the data type defined is $\mathbf{D}$
- The syntax $\mathbf{D}(\mathbf{A_1},\ldots,\mathbf{A_n})\; \rightarrow \mathbf{Z}$ conveys that a data type is an intial algebra.
- $\{\mathbf{A_1},\ldots,\mathbf{A_n}\}$ is the set of type parameters/type variables of the data type $\mathbf{D}$.
- $\mathbf{C_1},\ldots,\mathbf{C_n}$ are the constructors of the data type $\mathbf{D}$.
- $\tilde{\mathbf{P_i}}$ is the set of type parameters used in constructor $\mathbf{C_i}$ such that $\tilde{\mathbf{P_i}}$ is a subset of $\{\mathbf{A_1},\ldots,\mathbf{A_n}\}$.

$$\tilde{P_i} \;\subseteq\; (A_1,\ldots,A_n)$$

- The union of all the $\mathbf{P_i}$ for $\leq i \leq n$ should result in set $\{\mathbf{A_1},\ldots,\mathbf{A_n}\}$.

$$\bigcup_{i=1}^{n} \tilde{P_i} \;=\; \{A_1,\ldots,A_n\}$$

- $\mathbf{F_i}\ \tilde{\mathbf{P_i}}\ \mathbf{Z}$ is a type using the type variables $\mathbf{P_i}$ and $\mathbf{Z}$. $\mathbf{F_i}\ \tilde{\mathbf{P_i}}\ \mathbf{Z}$ is the input type of constructor $\mathbf{C_i}$.

Section 2.1 shows a few examples of data type definitions in MPL. The sequential MPL constructs that work on data types are described in Section 2.2.

2.1 Examples of Data Type

2.1.1 Boolean Data Type

The boolean data type can be defined in MPL as:

```
data Bool -> C = True  :: -> C
                 False :: -> C
```

The name of the data type is **Bool** and it has two constructors, **True** and **False**. **Bool** doesn't have any type variables. Both the constructors don't take any input and their output type is **Bool**.

2.1.2 Natural Number Data Type

The inductive defintion of natural numbers can be used to define the MPL data type for natural numbers called **Nat** as shown below:

```
data Nat -> C = Zero  ::   -> C
                Succ  :: C -> C
```

Nat data type is made up of two constructors, **Zero** and **Succ** meaning that all natural numbers can be expressed in terms of these data types. Constructor **Zero** doesn't take any input and its output type is **Nat**. Constructor **Succ** takes a natural number of type **Nat** as input and returns the next natural number of type **Nat** as the output. The **Succ** constructor is recursive.

2.1.3 List Data Type

The **List** data type can be defined as below:

```
data List(A) -> C = Nil   ::     -> C
                    Cons  :: A,C -> C
```

List data type is polymorphic in type variable **A** meaning that this data type can be used to store elements of any type **A** in a list. List is made up of two constructors, **Nil** and **Cons**. **Nil** constructor represents an empty list. **Nil** doesn't take any input and its output type is **List**(**A**). **Cons** constructor takes two inputs, an element of type **A** and a list of type **List**(**A**) and puts the element at the front of the list. The output of the **Cons** constructor is **List**(**A**). **Cons** constructor is recursive.

2.1.4 Either Data Type

Either data type is defined as below:

```
data Either(A,B) -> C = Left   :: A -> C
                        Right  :: B -> C
```

Either data type has two constructors **Left** and **Right**. This data type can be used as the output type of a function that returns two different types as output, say **A** and **B**. When the function returns the value of type **A**, **Left** constructor is used and when the function returns the value of type **B**, **Right** constructor is used. This data type is usually used to handle the output of a function that can fail for some inputs. Function returns an error message with the **Left** constructor and the normal output with the **Right** constructor.

2.2 Constructs for Data Types

There are some sequential MPL constructs that work with data types. They are case, constructor and fold constructs. They are described in this section.

2.2.1 constructor

A data type has a set of constructors. As the name suggests, the constructors of a data type are used to generate terms/instances of that data type.

For example, the constructor **Zero** and **Nat** can be used to generate any natural number:

$$Zero,\ Succ(Zero),\ Succ(Succ(Zero)),\ Succ(Succ(Zero))\dots$$

Zero represents 0, **Succ**(**Zero**) represents 1, **Succ**(**Succ**(**Zero**)) represents 2 etc.

The two constructors of **List**, **Nil** and **Cons** can be used to represent a polymorphic list. Consider a list of integers, then some examples of a list of integers created using the constructors of **List** data type are:

$$Nil,\ Cons(1, Nil),\ Cons(1, Cons(2, Nil)),\ Cons(1, Cons(2, Cons(3, Nil)))\dots$$

The can be equivalently written as below in the convenient syntax for lists:

$$[\,],\ [1],\ [1,2],\ [1,2,3]\dots$$

Nil or [ ] represents an empty list, $\mathbf{Cons(1, Nil)}$ or $[\mathbf{1}]$ represents a singleton list containing only integer 1 etc.

2.2.2 case Construct

case is a powerful MPL construct used with data types. Its syntax is:

$$\begin{array}{l} \mathbf{case\ \ t\ \ of} \\ \qquad \mathbf{C_1\ \ us_1\ \ \to t_1} \\ \qquad\qquad \vdots \\ \qquad \mathbf{C_n\ \ us_n\ \ \to t_n} \end{array}$$

First argument of the case construct is a sequential term represented in the syntax by **t**. The sequential term **sterm** evaluates to a data type, say **D** in this case.

case branches on the constructors of a data type (in this case, data type **D**) and selects a particular branch based on what the first argument **t** evaluates to. In the case syntax $\mathbf{C_1}, \dots, \mathbf{C_n}$ are the constructors of the data type **D**. $\mathbf{us_i}$ is the list of arguments (which are sequential terms) that the constructor $\mathbf{C_i}$ takes. Once a particular branch say $\mathbf{C_k}$ is selected, the term corresponding to the constructor, in this case $\mathbf{t_k}$ is executed.

Two examples that use case construct on data types, namely **Nat** and **List** data type, are discussed below.

Figure 1 shows an example of a function **convNatToInt** where **Nat** data type is cased on. The function **convNatToInt** takes the **Nat** representation as input and returns the corresponding integer representation. **convNatToInt** is a recursive function that returns the integer 0 for the **Zero** constructor. For the **Succ**(**n**) branch, the function adds 1 for the **Succ** constructor and calls itself recursively on argument **n**. The function is essentially counting the numbers of the **Succ** constructor in a **Nat** representation.

Figure 2 shows an example of a function that takes two lists as inputs and gives as output the appended list. The function cases on the first list. Note that instead of listing **Nil** and **Cons** in the two branches, the convenient syntax, [ ] for **Nil** and : for **Cons**, is used. Corresponding to the **Nil** branch, the second list is returned. In the **Cons** branch containing two arguments, the first argument of the **Cons** is added to the beginning of the list obtained as a result of appending the second argument of the **Cons** to the second list.

```
data Nat -> C = Zero ::   -> C
                Succ :: C -> C

-- convert Nat representation to Integer
convNatToInt :: Nat -> Int =
  nat -> case nat of
           Zero    -> 0
           Succ(n) -> 1 + convNatToInt(n)
```

Figure 1: Using case construct with Nat data type

```
data List(A) -> C = Nil  ::     -> C
                    Cons :: A,C -> C

-- append two lists
append :: List(A),List(A) -> List(A) =
  t1,t2 -> case t1 of
             []    -> t2
             x:xs  -> x:append(xs,t2)
```

Figure 2: Using case construct with List data type

2.2.3 fold construct

Recursion is an important tool when developing sequential MPL programs. fold is an MPL construct that allows programmers to use recursion in a disciplined manner. Functions written using fold construct belong to a set of functions called Higher Order Primitive Recursive Functions. Functions belonging to the set of Higher Order Primitive Recursive Functions have a property that they terminate, a useful property to have in a function.

The syntax of fold is:

$$\begin{array}{l} \textbf{fold}\ \ \mathbf{t}\ \ \textbf{of} \\ \qquad \mathbf{C_1}\ :\ \mathbf{us_1}\ \ \rightarrow \mathbf{t_1} \\ \qquad\qquad \vdots \\ \qquad \mathbf{C_n}\ :\ \mathbf{us_n}\ \ \rightarrow \mathbf{t_n} \end{array}$$

where term $\mathbf{t}$ is being folded over. The type of $\mathbf{t}$ is a data type. If the type of $\mathbf{t}$ is a data type $\mathbf{D}$ then $\mathbf{C_1}, \ldots, \mathbf{C_n}$ are the constructors of $\mathbf{D}$. $\mathbf{us_i}$ is the sequence of arguments and $\mathbf{t_i}$ is the term associated with the constructor $\mathbf{C_i}$.

Two examples have been discussed below that describe folding over **Nat** and **List** data types. The same examples were written with case construct using normal recursion in Figure 1 and 2.

Figure 3 shows a program that takes a natural number in the **Nat** representation and converts it to corresponding integer notation. The fold construct is used to write this program. The general type of the fold construct for **Nat** data type is:

$$foldNat\ ::\ C\ ,\ (C \rightarrow C)\ ,\ Nat\ \rightarrow\ C$$

The type of **foldNat** construct represents that it has three inputs: first argument is a function that handles the **Zero** constructor, second argument is a function that handles the **Succ** constructor and the third argument is the **Nat** data type being folded over. Since the **Zero** constructor doesn't take any input, the corresponding function is a constant function as seen by its type. If the output type expected as a result of folding over **Nat** is **C** then the functions corresponding to both the constructors should output a term of type **C**.

In the program in Figure 3, corresponding to the **Zero** case, integer 0 is returned. Corresponding to the **Succ** case, the **Nat** argument of **Succ** has been folded over and is already in its integer representation. All that remains to be done is to add 1 to the integer representation of the **Succ** argument.

Figure 4 shows a program that takes two lists and converts appends them. The fold construct is used to write this program. The general type of the fold construct for **List** data type is:

$$foldList\ ::\ C\ ,(A, C \rightarrow C)\ ,\ List(A)\ \rightarrow\ C$$

In the program in Figure 4, corresponding to the **Nil** constructor, the second list is returned. Corresponding to the **Cons** constructor, the first argument of **Cons** is consed infront of its second argument.

3 Codata Type and Constructs for Codata Type

MPL allows the programmers to define codata types, a facility lacking in most functional programming languages. A functional language with the facility of codata types is the programming

```
data Nat -> C = Zero ::   -> C
                Succ :: C -> C

-- convert Nat representation to Integer
convNatToInt:: Nat -> Int =
  nat -> fold nat of
           Zero     -> 0
           Succ : n -> 1 + n
```

Figure 3: folding over Nat data type

```
data List(A) -> C = Nil  ::     -> C
                    Cons :: A,C -> C

-- append two lists
append :: List(A),List(A) -> List(A) =
  t1,t2 -> fold t1 of
             Nil  :    -> t2
             Cons :x,r -> x:r
```

Figure 4: folding over List data type

language Charity by Cockett. Codata types provide means of representing potentially infinite structures that are evaluated lazily.

Syntax of the codata definition in MPL is:

$$\begin{aligned} \textbf{codata}\ \ \mathbf{Z} \ \rightarrow\ \mathbf{C}(\mathbf{A_1},\ldots,\mathbf{A_n}) \quad = \quad & \mathbf{D_1} : \mathbf{Z} \ \rightarrow\ \mathbf{F_1}\ \tilde{\mathbf{P_1}}\ \mathbf{Z} \\ & \vdots \\ & \mathbf{D_n} : \mathbf{Z} \ \rightarrow\ \mathbf{F_n}\ \tilde{\mathbf{P_n}}\ \mathbf{Z} \end{aligned}$$

Noticeable things about the codata type defintion are below:

- The name of the codata type defined is $\mathbf{C}$
- The syntax $\mathbf{Z} \ \rightarrow\ \mathbf{C}(\mathbf{A_1},\ldots,\mathbf{A_n})$ conveys that a codata type is a final algebra.
- $\{\mathbf{A_1},\ldots,\mathbf{A_n}\}$ is the set of type parameters/type variables of the codata type $\mathbf{C}$.
- $\mathbf{D_1},\ldots,\mathbf{D_n}$ are the destructors of the codata type $\mathbf{C}$.
- $\tilde{\mathbf{P_i}}$ is the set of type parameters used in destructor $\mathbf{D_i}$ such that $\tilde{\mathbf{P_i}}$ is a subset of $\{\mathbf{A_1},\ldots,\mathbf{A_n}\}$.

$$\tilde{P_i} \ \subseteq\ (A_1,\ldots,A_n)$$

- The union of all the P_i should result in set $\{A_1,\ldots,A_n\}$.

$$\bigcup_{i=1}^{n} \tilde{P_i} \ =\ \{A_1,\ldots,A_n\}$$

- $\mathbf{F_i}\ \tilde{\mathbf{P_i}}\ \mathbf{Z}$ is a type using the type variables $\mathbf{P_i}$ and $\mathbf{Z}$. $\mathbf{F_i}\ \tilde{\mathbf{P_i}}\ \mathbf{Z}$ is the output type of destructor $\mathbf{D_i}$.

3.1 Examples of Codata Type

3.1.1 InfList Codata Type

InfList codata type is defined as:

```
codata C -> InfList(A) = Head :: C -> A
                         Tail :: C -> C
```

As the name suggests **InfList**(**A**) data type can be used to store infinite list of things of type **A**. **InfList**(**A**) has two destructors, **Head** and **Tail**. **Head** destructor is a function that takes an infinite list of elements of type **A** and returns the first element of the infinite list. **Tail** destructor is a function that takes an infinite list of elements of type **A**, removes the first element and returns remainder of the list.

3.1.2 Lazy Triple

Triple can be defined in MPL using codata types as:

```
codata C -> Triple(D,E,F) = Proj0 :: C -> D
                            Proj1 :: C -> E
                            Proj2 :: C -> F
```

Triple has three destructors, **Proj0**, **Proj1**, **Proj2**. These destructors can then be used to get the first, second or the third element from the triple.

Since tuples/products are useful programming constructs to have in a language, MPL allows the programmers to use a **n**-tuple without defining the codata type for it. #**i** can used to project the $\mathbf{i^{th}}$ element (for $\mathbf{1} \leq \mathbf{n}$) from the **n**-tuple.

3.2 record and destructor Constructs

A record is used to hold value of a codata type. The record construct has the form:

$$(:\ \mathbf{D_1}\ \mathbf{us_1}\ :=\ \mathbf{t_1}, \ldots, \mathbf{D_n}\ \mathbf{us_n}\ :=\ \mathbf{t_n}\ :)$$

Here, (: :) is the special synatx for records. $\mathbf{D_i}$ is the $\mathbf{i^{th}}$ destructor of the codata type, the record of which is being formed and $\mathbf{us_i}$ is the sequence of arguments corresponding to the destructor.

The following record creates an record of **Triple** codata type:

```
(: Proj0 := 1 , Proj1 := 2.3 , Proj2 := ‘‘Hello’’ :)
```

Destructors retrieve value from a record. The MPL term

```
Proj2 (: Proj0 := 1 , Proj1 := 2.3 , Proj2 := ‘‘Hello’’ :)
```

will evaluate to **“Hello”**.

The above triple can directly be written in MPL without defining the **Triple** codata type as below:

```
(1, 2.3, ‘‘Hello’’)
```

The MPL term `#3 (1, 2.3, ‘‘Hello’’)` will evaluate to `‘‘Hello’’`.

3.3 unfold construct

unfold construct is a constructive operation for codata types. Codata types are built using unfold. Syntax of unfold is:

$$\begin{array}{l} \textbf{unfold} \\ \quad \mathbf{s} \ \Rightarrow \\ \qquad \mathbf{D_1} \ : \ \mathbf{us_1} \ \rightarrow \ \mathbf{t_1} \\ \qquad \qquad \vdots \\ \qquad \mathbf{D_n} \ : \ \mathbf{us_n} \ \rightarrow \ \mathbf{t_n} \\ \textbf{on} \ \ \mathbf{t} \end{array}$$

- $\mathbf{D_1}, \ldots, \mathbf{D_n}$ are destructors of a codata type.
- $\mathbf{us_i}$ is the sequence of arguments corresponding to the destructor $\mathbf{D_i}$ and $\mathbf{t_i}$ is the corresponding term.
- The set of terms corresponding to the different destructors are also known as the threads of the unfold.
- $\mathbf{s}$ is a state accessible to all the destructors $\mathbf{D_1}, \ldots, \mathbf{D_n}$.
- $\mathbf{t}$ is initial value that is to be unfolded.

In Figure 5, unfold construct is used to generate an infinite list of natural numbers starting from 0. The initial value supplied to the unfold is 0. The thread corresponding to the **Head** destructor generates the current state and the thread corresponding to the **Tail** destructor generates the next state.

```
codata C -> InfList(A) = Head :: C -> A
                         Tail :: C -> C


-- infinite list of numbers starting from 0
genNat :: () -> InfList(A) =
   -> unfold
        n =>
          Head : -> n
          Tail : -> n + 1
      on 0
```

Figure 5: Using unfold to generate an infinite list of numbers

4 Non Data/Codata Constructs

These are the MPL constructs that are not used with either the data types or the codata types. They are function call, where, variable, constant and if-then-else constructs.

4.1 function call construct

MPL allows the programmer to define their own functions. It also has inbuilt functions such as addition, subtraction, multiplication, divison, equality testing etc. Both the classes of functions can be called in MPL functions or processes. Syntax of a function call is:

$$\mathbf{fname}\ \ (\mathbf{arg_1}, \ldots, \mathbf{arg_n})$$

where **fname** is the function (defined/builtin) being called with **n** arguments $\mathbf{arg_1}, \ldots, \mathbf{arg_n}$. The arguments

The function should be called with the right number of parameters otherwise a semantic error is raised.

4.2 if-then-else construct

if-then-else is a sequential construct that takes three arguments. It syntax is:

$$\mathbf{if}\ \ \textit{boolean}\ \ \mathbf{then}\ \ \textit{term}_1\ \ \mathbf{else}\ \ \textit{term}_2$$

where the first argument is a sequential MPL construct that evaluates to a boolean value. If the boolean value evaluates to **true** then $\mathbf{term_1}$ is executed else $\mathbf{term_2}$ is executed.

4.3 variable construct

variables construct allows MPL programs to use variable names in the body of an MPL program. The variable names used in the program should be introduced either through a variable pattern or through a local variable definition through a where construct. A variable name used inside an MPL program without being introduced legally results in a semantic error.

4.4 constant construct

Constants which can be integers, floats, characters, or strings can be used in the body of an MPL program.

4.5 where construct

where is a sequential MPL construct used to define:

- local functions that are needed only inside the body of a given function, and
- local variables that can be used in the body of the function.

Figure 6 shows an example of an MPL function that uses the where construct. The function **fahToKel** function takes temperature in degree Fahrenheit and converts it to the temperature in Kelvin. This conversion is achieved by first converting degree Fahrenheit to degree Celsius and then converting degree Celsius to Kelvin.

Function **fahToCel** is a local function defined using the where construct which converts degree Fahrenheit to degree Celsius using the relation $^{\circ}C = (5 * {^{\circ}F} - 160)/9$ where $^{\circ}C$ and $^{\circ}F$ are the temperature in Celsius and Fahrenheit respectively. The function **fahToCel** is visible only inside the body of function **fahToKel**.

The calculation to convert degree Celsius to Kelvin is stored in variable **kel**. The relation between Celsius and kelvin is $K = {}^{\circ}C + 273$ where K is the tempearture in Kelvin. The local variable **kel** can then be used inside the main body of the function **fahToCel**. Thus, using the where construct a seemingly compilcated looking program can be broken down into simpler components and can be cleanly implemented.

```
-- convert degree Fahrenheit to Kelvin
fun fahToKel :: Double -> Double =
  fah -> kel
      where
        kel = fahToCel(fah) + 273.15
        fahToCel :: Double -> Double =
            fah -> (5*fah - 160)/9
```

Figure 6: using where construct

4.6 switch construct

The switch construct is of the form:

$$\begin{array}{lll}\textbf{switch} & & \\ \quad \mathbf{p_1} & = & \mathbf{t_1} \\ & \vdots & \\ \quad \mathbf{p_n} & = & \mathbf{t_n} \\ \quad \textbf{default} & = & \mathbf{t_{def}}\end{array}$$

switch consists of a list of pairs of terms. Each pair $(\mathbf{p_i}, \mathbf{t_i})$ corresponds to a line in the body of the switch construct. All the first elements of the list of pairs must return a boolean value. The first term from $\mathbf{p_1}, \ldots, \mathbf{p_n}$ to returns a **True** will have its corresponding term executed. If none of $\mathbf{p_1}, \ldots, \mathbf{p_n}$ return a true then the term $\mathbf{t_{def}}$ corresponding to the **default** case is executed.

Figure 7 shows an example where switch construct is used. Function **monthName** takes an integer as input. If the integer is between one and twelve it returns a string which is the name of the corresponding month otherwise it returns a string saying that the month number is invalid.

5 Pattern Matching

Like most functional programming languages, for example- Haskell, OCaml, ML etc, sequential MPL supports pattern matching. An abstraction

$$v_1, \ldots, v_n \quad \to t$$

```
-- convert month number to month name
fun monthName :: Int -> String =
  month -> switch
             month == 1 = ‘‘January’’
             month == 2 = ‘‘February’’
             month == 3 = ‘‘March’’
             month == 4 = ‘‘April’’
             month == 5 = ‘‘May’’
             month == 6 = ‘‘June’’
             month == 7 = ‘‘July’’
             month == 8 = ‘‘August’’
             month == 9 = ‘‘September’’
             month == 10 = ‘‘October’’
             month == 11 = ‘‘November’’
             month == 12 = ‘‘December’’
             default     = ‘‘Invalid month number’’
```

Figure 7: using switch construct

generalizes to the patterned abstraction

$$\begin{array}{ll} p_{1,1}, \ldots, p_{1,n} & \to t_1 \\ \vdots & \\ p_{m,1}, \ldots, p_{m,n} & \to t_m \end{array}$$

which is a list of pattern-term phrase. A pattern-term phrase consists of a list of patterns and a sequential term corresponding to it. When using pattern matching, term $\mathbf{t_i}$ is executed only if $\mathbf{p_{i,1}}, \ldots, \mathbf{p_{i,n}}$ is the first pattern list which matches the input.

Variables, constructors, records and products can be used as patterns giving variable patterns, constructor patterns, record patterns and product patterns respectively. Sometimes, elements of a pattern aren't used in the corresponding term of the pattern-term phrase. A special kind of pattern called the don't care pattern represented by an underscore (‘‘_’’) is used in such a case.

Pattern matching is compiled to case statements. The compilation of pattern matching to case has been described in Chapter.

Figure seqMPL:pattMatchExample gives a few examples of the use of pattern matching in MPL functions. Function **isZero** takes an input of type **Nat** and returns **True** if the input is **Zero** and **False** otherwise. **isZero** uses a constructor pattern in the first pattern-term phrase and a don't care pattern in the second pattern-term phrase.

or function in Figure 8 takes a pair of booleans and returns **True** if either of the elements of the pair is **True** and **False** otherwise. **or** function definition uses different kinds of patterns for pattern matching: product patterns, constructor patterns and don't care patterns. An interesting things to note here is that constructor and don't care patterns have been used inside of a product pattern. A pattern that contains other patterns is called a nested pattern.

thirdProj function in Figure 8 takes a record pattern of type $\mathbf{Triple(D, E, F)}$ and returns the element corresponding to the third destructor from the pattern.

```
-- check if the natural number is zero
isZero :: Nat -> Bool =
  Zero -> True
  _    -> False


-- boolean or function
or :: (Bool,Bool) -> Bool =
  (True,_) -> True
  (_,True) -> True
  _        -> False

-- third projection function
thirdProj :: Triple(D,E,F) -> F =
  (:Proj0 := x, Proj1 := y, Proj2 := z :) -> z
```

Figure 8: Examples of pattern matching

6 Higher Order Functions in MPL

Higher order functions can defined in MPL using a simple higher order codata data type called **Exp** (exponential) defined as:

```
codata C -> Exp(A,B) = App : A,C -> B
```

The type of the destructor **App** (function application) is:

$$\mathbf{A},\ \mathbf{Exp(A, B)}\ \rightarrow\ \mathbf{B}$$

$\mathbf{Exp(A, B)}$ is the type of total functions from type $\mathbf{A}$ to type $\mathbf{B}$ and is also written as $\mathbf{A}\ \Rightarrow \mathbf{B}$ or $\mathbf{B^A}$ in literature. An example that uses the exponential codata type to implement a function $\lambda\mathbf{x.x+1}$ is:

```
fun lambdaFun  =
   ->  (: App(x) :=  x + 1 :)
```

Here, function **lambdaFun** form a record of **Exp** codata type which can then be destructed using the **App** destructor presumably inside the body of another function. The term $\mathbf{App\ (5, lambdaFun())}$ will evaluate to 6. Thus **Exp** data type provides a mechanism to pass a function as input to other functions.