Type Inferencing in MPL

March 23, 2017

1 Structure of an MPL Program

An MPL program consists of comments (single line comments, multiline comments), data and co-data definitions, protocols and coprotocol definitions and a definition block.

Indentation rather than bracketing is used to structure a block of text in MPL. This makes

1.1 comments in MPL

Single line comments are represented by "–". Multiline comments are things between "-" and "-".

1.2 data definitions

MPL allows programmers to define data type. For example ,

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

The letter C in the above data definition can be thought of as a placeholder for Bool. However, the motivation for the notation **Bool** $\rightarrow$ **C** goes depper than this and has been borrowed from tne notation of algebra from Category Theorey. True and False can be considered as two constructors The above data definition can be read like this.

Bool is a data type having two constructors True and False. True constructor doesn't take any input and produces an output of type Bool.

Lets try this for a slightly more complicated data type List.

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

List data type has a type variable A, i.e this is defining a polymorphic list and not a list of some specific type. This data type can be used to define a list of integers, list of characters, list of strings, list of lists etc. The type of the Cons constructor can be read as this. Cons takes two arguments of type A and List (A) and outputs a thing of type List(A).

1.3 codata definitions

MPL allows coinductive data definitions. A coindcutive data type is defined in MPL with the following syntax.

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

Head and Tail are called the destuctors of the Codata type Stream. C -¿ Stream part of the syntax in the definition comes from Co-Algebras. C can be considered as a place holder for Stream(A). The type of the **Head** destructor can be read as a function taking Stream(A) type as input and generating a thing of type A as output. Similarly, Tail takes Stream(A) as input and returns a thing of type Stream(A) as output (by removing the first element of the Stream).

1.4 function definitions

The below function definition shows the syntax of normal function definition in MPL.

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

1. Function definitions are proceeded by the by the keyword fun followed by the function name.
2. :: seprates the function name from its expected/annotated type. The programmer annotates the function with the type he expects the function to be. Notice that this may/maynot be the most general type of the function. The annotated type may be a more specific type if the programmer doesn't wish to use the generic version.

   In the defintion above [A] is the syntactic sugar for List (A). The input types of the functions are on the left side of the arrow and are separated by ,.