Typing Rules for Sequential MPL

August 6, 2017

1 Type Inferencing

1.1 Introduction

Once an MPL program is lexed and parsed, an abstract syntax tree (AST) for the program is generated. An AST syntax is a faithful representation of the original MPL program in that all the information present in the original program is present in the AST. The next stage in the interpretation of the MPL program is type inferencing. This step ensures that only meaninful programs that have valid types constinue the process of interpretation.

This chapter deals with the type inference of the various MPL constructs like functions, terms, patterns, pattern phrases, processes and process commands.

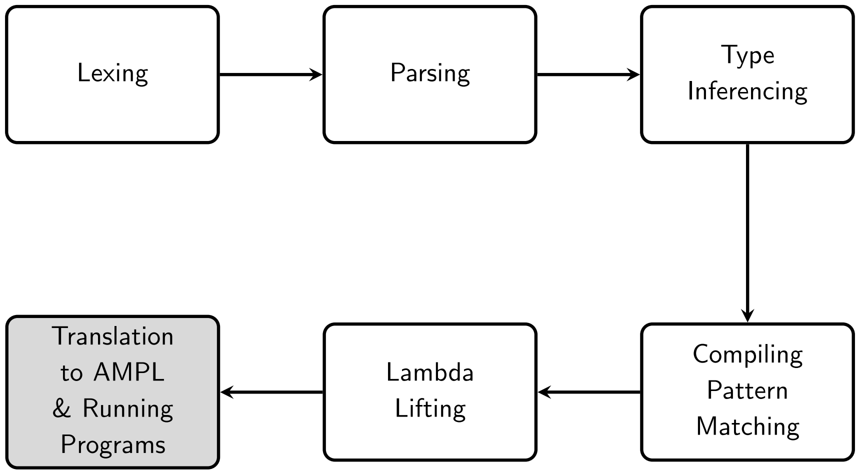
Lexing
Parsing
Type Inferencing
Compiling Pattern Matching
Lambda Lifting
Translation to AMPL & Running Programs

Figure 1: Interpretation Stages of MPL

1.2 Type Formation Rules

Let T be a set of atomic types, Ω_D be a set of type formation symbols for data types and ω_D is a function called arity that describes the number of parameters that the type needs.

$$\omega_D \ : \ \Omega_D \ \rightarrow \ \mathbb{N}$$

Similarly, ω_C is the arity function for Ω_C which is a set of type formation symbols for codata types.

$$\omega_C \ : \ \Omega_C \ \rightarrow \ \mathbb{N}$$

$$\frac{A \in T}{A \quad type}$$

$$\frac{A_1\ type \qquad \dots \qquad A_n\ type}{A_1 \times \dots \times A_n \quad type}\ prod$$

$$\frac{A_1\ type \qquad \dots \qquad A_n\ type \qquad \omega(D)\ =\ n}{D\ (A_1 \dots A_n) \quad type}\ data\ type$$

$$\frac{A_1\ type \qquad \dots \qquad A_n\ type \qquad \omega(C)\ =\ n}{C\ (A_1 \dots A_n) \quad type}\ codata\ type$$

$$\frac{A_1\ type \qquad \dots \qquad A_n\ type \qquad \omega(P)\ =\ n}{P\ (A_1 \dots A_n) \quad type}\ protocol$$

$$\frac{A_1\ type \qquad \dots \qquad A_n\ type \qquad \omega(CP)\ =\ n}{CP\ (A_1 \dots A_n) \quad type}\ coprotocol$$

Table 1: Type Formation Rules

Similarly, ω_P is the arity function for Ω_P and ω_{CP} is the arity function for ω_{CP}. Ω_P and Ω_{CP} are respectively the set of type formation rules for protocols and coprotocols respectively.

Elements of T can be type variables or type constants like Int, Double and Char. Apart from the atomic types, MPL also has data types, codata types and products as types. Type formation rules describes the rules for the construction of valid types from the elements of T.

The type formation rules for MPL are given by the rules in Table 1.

1.3 Type Inferencing

Consider an Algebraic Type System consisting of functions and variables as terms. The term formation rules for the terms are as below.

$$\frac{}{\Gamma, x : A \vdash x : A}\ \text{var}$$

$$\frac{\Gamma \vdash m_1 : A_1, \qquad \dots \qquad \Gamma \vdash m_k : A_k \qquad f : A_1, \dots, A_k \to B \in \Omega}{\Gamma \vdash\ f(m_1, \dots, m_n) : B}\ \text{fun}$$

In this type system consider a function,$f(x, y)\ =\ x$ such that $f : A, B \to A \in \Omega$. Let us try inferring the type of $f\ (1, y)$.

$$\frac{y : B \vdash y : B \qquad y : B \vdash 1 : Int \qquad f : A, B \to A \in \Omega}{y : B \vdash \; f(1, y) : Int}$$

Please note that $f : Int, B \to Int$ is a more specific type than the type of f initially assigned in Ω. The new more specific type of f is consistent with its original more generic type. However, this way of type inferencing while suitable for calculation by hand doesn't lend itself well to automation. For this reason, the typing rules are enhanced with equations that capture the constraint relationship between the different parts of the construct being type inferred.

$$\frac{}{\Gamma, x : A \vdash x : B \;\; \langle A = B \rangle}\;\text{var}$$

$$\frac{\Gamma \vdash m_1 : A'_1, \qquad \ldots \qquad \Gamma \vdash m_k : A'_k \qquad f : A_1, \ldots, A_k \to B \in \Omega}{\Gamma \vdash \; f(m_1, \ldots, m_n) : C \;\; \langle B = C, A_1 = A'_1, \ldots, A_k = A'_k \rangle}\;\text{fun}$$

Now consider type inferring the function $f(1, y)$ in this setup.

$$\frac{y : B \vdash y : B' \qquad y : B \vdash 1 : Int \qquad f : A', B' \to A \in \Omega}{y : B \vdash \; f(1, y) : Int \quad \langle A' = Int, \; B' = B, \; A = Int \rangle}$$

Once the type equations are solved by relacing the value of A, B and A' in function type of f, $Int, B \to Int$ is obtained. This is the type of function f.

The type inference has been broken down into two mechanical steps, namely *Generation of Type Equations* and *Solving Type Equations.* The mechanical and algorithmic nature of this presentation is useful when implementing type systems on a computer.

One drawback of the equational presentation of the type systems mentioned in the previous paragraph is that one ends up with a big linear list of equations. Solving this list of equations maybe slow as one needs to go through the entire list every time in order to look for substitutions.

In this thesis, we have modified this presentation to take advantage of the structure of the *type inference tree* when searching for substitutions. In this approach, the algorithm only searches the part of the type inference tree which could have generated these substitutions. Thus our algorithm localises the substitution search leading to faster soltuion of type equations. Our algorithm also has an added advantage of localising the type errors. This results in better location accuracy when reporting the type errors.

Consider a function $f(a, b)$ such that $f : \; A, B \to C \in \Omega$ and one needs to type infer $f(x + y, z)$ in our setup. In the below *type inference tree*, the context x:X,y:Y,z:Z has been replaced with symbol Γ for the purpose of succinct representation.

$$\frac{\dfrac{\dfrac{}{\Gamma \vdash x : D} \qquad \dfrac{}{\Gamma \vdash y : E}}{\Gamma \vdash x + y \; : C} \qquad \dfrac{}{\Gamma \vdash z : B} \qquad f : P \;, Q \to R \in \Omega}{\Gamma \vdash \; f(x + y, z) : A \quad \langle A' = Int, \; B' = B, \; A = Int \rangle}$$

1.4 Overview of Type Inference of MPL Constructs

Type inference involves two main steps.

- **Generating Type Equations** - Type Equations represent the constraint relationship between the different parts of the construct being type inferred.

  Using the typing rules, a series of equations are generated in the proof search direction. The equation structure hierarchical following the shape of the type inference tree. Thus, the placement of an equation inside a type equation depends on thes the location within a type inference tree that resulted in that equation within the type equation. This approach allows hierarchical solution of type equations and better localisation of type errors.

- **Solving Type Equations** - The type equations are solved in order to get the most general type of a given program.

In this chapter, the generation fo type equations for the Sequential MPL (terms and functions) is dicussed followed by the generation of type equation for Concurrent MPL. Finally, we look at an algorithm to solve the type equations to get the most general type. Section discusses type equation generation for sequential terms, Section discusses type equation generation for pattern

2 Generating Type Equations for Sequential Terms

Here the type inference rules for the various sequential MPL constructs are dicussed. The type inference rules can be thought of as term formation rules enhanced with the typing information of the constituting terms.

2.1 Data Type Terms (case, constructor and fold terms)

case, constructor and fold are the sequential MPL terms that work on data types. Section 2.1.1 describes a data type declaration in MPL and Section describes the typing rules for case, constructor and fold terms.

2.1.1 Data Type Declaration

Consider a pair of mutually recursive data types: data type $D(A_1, \dots, A_k)$ with constructors $C_1, \dots, C_m$ and data type $P(B_1, \dots, B_l)$ with constructors $Q_1, \dots, Q_r$ defined below:

$$\begin{array}{lll}
\multicolumn{3}{l}{\mathit{data}} \\
\quad D(A_1, \dots, A_k) \ \rightarrow \ Z = & & \\
\qquad C_1 : T_{11}, \dots, T_{1a} & \rightarrow & Z \\
\qquad \vdots \qquad \vdots & & \vdots \\
\qquad C_m : T_{m1}, \dots, T_{mn} & \rightarrow & Z \\
\quad \mathit{and} & & \\
\quad P(B_1, \dots, B_l) \ \rightarrow \ Y = & & \\
\qquad Q_1 : T'_{11}, \dots, T'_{1b} & \rightarrow & Y \\
\qquad \vdots \qquad \vdots & & \vdots \\
\qquad Q_r : T'_{r1}, \dots, T'_{rs} & \rightarrow & Y
\end{array}$$

Following properties are explained with the data type $D(A_1, \ldots, A_k)$ in mind. However, the same properties will hold true for the data type $D(B_1, \ldots, B_l)$ as well.

- The data type $D(A_1, \ldots, A_k)$ is polymorphic in type variables $\{A_1, \ldots, A_k\}$. Z is called a state variable. The union of all the type variables used in the different constructors of the data type is $\{Z, A_1, \ldots, A_k\}$.
- Suppose Constructor C_i takes j number of input terms.The types of these terms are represented by $T_{11}, \ldots, T_{1a}$ respectively.
- T_{ij} is a type expression such that:

$$Type\ Var\ (T_{ij}) \quad \subseteq \quad \{Z, A_1, \ldots, A_k\}$$

- For the constructor C_i of data type D:
  - the type of the fold function corresponding to that constructor is directly given by the type expression corresponding to the constructor in the data definition. The type of fold corresponding to constructor C_i is represented by the notation:

$$\forall\ Z, V_1, \ldots, V_x\ .\ T_{i1}, \ldots, T_{ij}\ \rightarrow\ Z \tag{1}$$

    where $\forall\ Z, V_1, \ldots, V_x$ represents that the the type expression $T_{i1}, \ldots, T_{ij}\ \rightarrow\ Z$, which is a function type, is universally quantified by the type variables $Z, V_1, \ldots, V_x$. In other words, the set of type variables $\{Z, V_1, \ldots, V_x\}$ is present in the body of the the type expression $T_{i1}, \ldots, T_{ij}\ \rightarrow\ Z$. $\{Z, V_1, \ldots, V_x\}$ is a subset of $\{Z, A_1, \ldots, A_k\}$, the set of all the type variables used in the data definition D.
  - the type of the constructor C_i is given by substituting the state variable Z in the body of the corresponding type expression with the name of the data definition along with its polymorphic variables $D(A_1, \ldots, A_k)$. This is represented by the notation:

$$(\forall\ Z, A_1, \ldots, A_k\ .\ T_{i1}, \ldots, T_{ij}\ \rightarrow\ Z)\ D(A_1, \ldots, A_k)$$

    Note that for this function, the complete set of type variables present in the data definition are put under universal quantification. This is done because the substitution of Z with $D(A_1, \ldots, A_k)$ is anyways going to have that effect. Rather than refreshing the variables under universal quantification after the substitution of Z with $D(A_1, \ldots, A_k)$, one might as well start with all the variables in the data definiton. The subtitution of Z with $D(A_1, \ldots, A_k)$ in the body of type expression corresponding to the constructor C_i can be represented by the notation:

$$(\forall\ A_1, \ldots, A_k\ .\ T_{i1}, \ldots, T_{ij}\ \rightarrow\ Z)\ \Big[D(A_1, \ldots, A_k)/Z\Big] \tag{2}$$

For all the constructors of a data type, the type of their folds and the types of constructors themselves, represented by (1) and (2) respectively, are inserted into the symbol table. Symbol table is a data structure used in the type inference process which acts as a repository of information for the different symbols used in the program.

The fold and constructor function types corresponding to the constructors of the data type definiton can be looked up and used for generating equations. However, before using the fold and the constructor function types in the generation of type equations, they must be α-renamed with fresh

$$\frac{\Gamma \vdash t_1 : T_1 \quad \langle E_1 \rangle \qquad \dots \qquad \Gamma \vdash t_j : T_j \quad \langle E_j \rangle}{} \text{cons}$$

Table 2: Typing Rule for constructor

variables to avoid any naming conflicts with variables already present in the type equation. For the same reason, the α-renaming technique is used with other sequential MPL constructs as well.

The below notation represents that the fold type of constructor C_i given by (1) has been substituted with fresh variables $V'_1, \dots, V'_x$ for $V_1, \dots, V_x$ respectively:

$$(T_{i1}, \dots, T_{ij} \to Z) \ \ [Z'/Z, V'_1/V_1, \dots, V'_x/V_x]$$

Similarly, the type of a constructor function given by (2) substituted with fresh variables $A'_1, \dots, A'_k$ for $A_1, \dots, A_k$ respectively can be represented as:

$$(T_{i1}, \dots, T_{ij} \to Z) \ [D(A_1, \dots, A_k)/Z \ , \ A'_1/A_1, \dots, A'_k/A_k]$$

2.1.2 Constructor Term

A data type has a set of constructors. The constructors of a data type sre used to generate instances of that data type.

The type inference rule for constructors is described in Table 2. The data type $D(A_1, \dots, A_k)$ referred to in the table has been defined in Section 2.1.1. Constructor C_i used in Table 2 is a constructor of data type $D(A_1, \dots, A_k)$ and it takes j arguments. A few noteworthy points about the type inference rule for the constructor C_i of data type D, provided in Table 2, are as follows:

- The output type of a constructor term is the data type of which it is a constructor. The equation corresponding to this relationship is:

  $$T \ = \ D(A'_1, \dots, A'_k)$$

  where $A'_1, \dots, A'_k$ are fresh variables used instead of the variables $A_1, \dots, A_k$ respectively.

- The types of the various arguments of the constructor are the type expressions at the corresponding positions of that data constructor in the data definition with the state variable Z substituted with $D(A'_1, \dots, A'_k)$. The type T_j of the j^{th} argument of constructor C_i is given by:

  $$T_j = T_{ij} \ \left[D(A_1, \dots, A_k)/Z \ , \ A'_1/A_1, \dots, A'_k/A_k\right]$$

  where T_{ij} is the type expression corresponding to the j^{th} argument of the constructor C_i in the data definition of D. Syntax $T_{ij} \ \left[D(A_1, \dots, A_k)/Z \ , \ A'_1/A_1, \dots, A'_k/A_k\right]$ represents that in the body of type expression T_{ij}, a list of substitutions $\left[D(A_1, \dots, A_k)/Z \ , \ A'_1/A_1, \dots, A'_k/A_k\right]$ are used.

- E_1, ..., E_j are the type equations generated for the terms t_1, ..., t_j respectively.

2.1.3 case Term

case term branches on the different constructors of a data type. Every branch consists of a constructor and a sequential term. When a branch with a constructor C_i is selected, the corresponding sequential term t_i is executed.

Table 3 describes the type inference rules for case term. Noteworthy points about the type inference rule for case term provided in Table 3 are below:

- The type of the term being cased on should be a data type. This should be the same data type of which $C_1, \ldots, C_m$ are constructors. This relationship is represented by the equation:

$$T_0 = D(A'_1, \ldots, A'_k)$$

  where $A'_1, \ldots, A'_k$ are fresh variables used instead of the variables $A_1, \ldots, A_k$ respectively.

- The output type of the case term is the type of the terms $t_1, \ldots, t_m$. Thus, in a well formed case expression the types of all the terms $t_1, \ldots, t_m$ should be the same.

- The type F_{ij} of the term x_{ij}, which is the j^{th} argument of the constructor C_i, is given by the following equation:

$$F_{ij} \ = \ T_{ij} \left[D(A_1, \ldots, A_k)/Z \ , \ A'_1/A_1, \ldots, A'_k/A_k \right]$$

  where T_{ij} is the type expression corresponding to x_{ij} in the data definition D. $\left[D(A_1, \ldots, A_k)/Z \ , \ A'_1/A_1, \ldots, A'_k/A_k \right]$ represents a list of substitutions to be made inside the body of the type expression T_{ij}.

- $E_1, \ldots, E_m$ are respectively the type equations generated for the terms $t_1, \ldots, t_m$ in the context expanded by the variables associated with the respective constructors.

- E_0 represents the set of type equations for term t, the term which is cased on.

2.1.4 fold Term

fold is used to implement recursion in a disciplined manner. The recursive functions written using fold always terminate. The function type corresponding to fold for a constructor is directly obtained from the type expression corresponding to the constructor in the data definition.

Before moving on the type inference of the fold term, one must remeber from the chapter on sequential MPL that folding over the data type D which is mutually recursive to data type P requires all the constructors of both the data types D and P listed in the fold term.

Table 4 describes the type inference rule for the fold term over data type D. Noteworthy points about the type inference rules for fold term from Table 4 are as follows:

- The type of the term being folded over should be a data type. This should be the same data type of which $C_1, \ldots, C_m$ are constructors. This relationship is represented by the equation:

$$T_0 = D(A'_1, \ldots, A'_k)$$

- The output type of the case term is the type of the terms $t_1, \ldots, t_n$. Thus, in a well formed fold expression the types of all these terms $t_1, \ldots, t_n$ should be the same.

$$\frac{\Gamma \vdash t : T_0 \ \langle E_0\rangle \qquad \begin{array}{lcl} \Gamma, x_{m1}:F_{m1},\ldots,x_{mn}:F_{mn} & \vdash & t_m : T_m \quad \langle E_m\rangle \\ \vdots & & \\ \Gamma, x_{11}:F_{11},\ldots,x_{1a}:F_{1a} & \vdash & t_1 : T_1 \quad \langle E_1\rangle \end{array}}{\Gamma \vdash \begin{array}{l} \mathsf{case}\ t \\ \mathsf{of} \left| \begin{array}{lcl} C_1\ x_{11},\ldots,x_{1a} & \rightarrow & t_1 \\ \vdots & \vdots & \vdots \\ C_m\ x_{m1},\ldots,x_{mn} & \rightarrow & t_m \end{array}\right. \end{array} : T} \text{ case}$$

$$\left\langle \exists \begin{array}{l} T_0, T_1, \ldots, T_m, \\ F_{11}, \ldots, F_{1a}, \\ \quad\vdots \\ F_{m1}, \ldots, F_{mn}, \\ A'_1, \ldots, A'_k \end{array} . \begin{array}{l} T_0 = D(A'_1, \ldots, A'_k), \\ T_1 = T, \ldots, T_m = T \\ F_{11} = T_{11}\left[D(A_1,\ldots,A_k)/Z\ ,\ A'_1/A_1,\ldots,A'_k/A_k\right], \\ \qquad\vdots \\ F_{1a} = T_{1a}\left[D(A_1,\ldots,A_k)/Z\ ,\ A'_1/A_1,\ldots,A'_k/A_k\right], \\ \qquad\vdots \\ F_{m1} = T_{m1}\left[D(A_1,\ldots,A_k)/Z\ ,\ A'_1/A_1,\ldots,A'_k/A_k\right], \\ \qquad\vdots \\ F_{mn} = T_{mn}\left[D(A_1,\ldots,A_k)/Z\ ,\ A'_1/A_1,\ldots,A'_k/A_k\right], \\ E_0, E_1, E_2, \ldots, E_m \end{array} \right\rangle$$

Table 3: Typing Rule for case term

$$
\frac{
t : T_0 \ \langle E_0 \rangle \qquad
\begin{array}{lcll}
x_{11} : F_{11}, \ldots, x_{1a} : F_{1a} & \vdash & t_1 : T_1 & \langle E_1 \rangle \\
\vdots & \vdots & \vdots & \\
x_{m1} : F_{m1}, \ldots, x_{mn} : F_{mn} & \vdash & t_m : T_m & \langle E_m \rangle \\
y_{11} : G_{11}, \ldots, y_{1a} : G_{1b} & \vdash & u_1 : U_1 & \langle E'_1 \rangle \\
\vdots & \vdots & \vdots & \\
y_{r1} : G_{r1}, \ldots, y_{rs} : G_{rs} & \vdash & u_r : U_r & \langle E'_r \rangle
\end{array}
}{
\Gamma \vdash
\begin{array}{l}
\mathsf{fold}\ t \\
\mathsf{of}
\left|
\begin{array}{lcl}
C_1 \ : \ x_{11}, \ldots, x_{1a} & \to & t_1 \\
\vdots & \vdots & \vdots \\
C_m \ : \ x_{m1}, \ldots, x_{mn} & \to & t_m \\
Q_1 \ : \ y_{11}, \ldots, y_{1b} & \to & u_1 \\
\vdots & \vdots & \vdots \\
Q_r \ : \ y_{r1}, \ldots, y_{rs} & \to & u_r
\end{array}
\right.
\end{array}
\quad : T
\quad
\left\langle \exists
\begin{array}{l}
T_0, T_1, \ldots, T_m, \\
F_{11}, \ldots, F_{1a}, \\
\quad \vdots \\
F_{m1}, \ldots, F_{mn}, \\
A'_1, \ldots, A'_k
\end{array}
.
\begin{array}{l}
T_0 = D(A'_1, \ldots, A'_k), \\
T_1 = T, \ldots, T_m = T \\
F_{11} = T_{11} \left[Z'/Z \ , \ A'_1/A_1, \ldots, A'_k/A_k \right], \\
\qquad \vdots \\
F_{1a} = T_{1a} \left[Z'/Z \ , \ A'_1/A_1, \ldots, A'_k/A_k \right], \\
\qquad \vdots \\
F_{m1} = T_{m1} \left[Z'/Z \ , \ A'_1/A_1, \ldots, A'_k/A_k \right], \\
\qquad \vdots \\
F_{mn} = T_{mn} \left[Z'/Z \ , \ A'_1/A_1, \ldots, A'_k/A_k \right], \\
E_0, E_1, \ldots, E_m, \\
E'_1, \ldots, E'_r
\end{array}
\right\rangle
}
\ \text{fold}
$$

Table 4: Typing Rule for fold term

- The type F_{ij} of the term x_{ij}, which is the j^{th} argument of the constructor C_i, is given by the following equation:

$$F_{ij} \; = \; T_{ij} \left[Z'/Z \; , \; A'_1/A_1, \ldots, A'_k/A_k \right]$$

  where T_{ij} is the type expression corresponding to x_{ij} in the data definition D.

- $E_1, \ldots, E_m$ are respectively the type equations generated for the terms $t_1, \ldots, t_m$ in the context expanded by the variables associated with the respective constructors in the fold term.

- E_0 represents the term t which is being folded over.

2.2 Codata Type Terms (record, product, destructor and unfold Terms)

record, product, destructor and unfold are the sequential MPL terms that work on codata types. Section describes a codata type declaration in MPL and Sections through respectively describe the typing rules for record, product, destructor and unfold term.

2.3 Codata Type Declaration

Consider a codata type $C(A_1, \ldots, A_k)$ having destructors $D_1, \ldots, D_m$ defined below:

$$\begin{array}{lllll} \textit{codata } Z \; \rightarrow \; C(A_1, \ldots, A_k) = & D_1 : T_{11}, \ldots, & T_{1a}, \; Z & \rightarrow & P_1 \\ & \vdots & \vdots & \vdots & \\ & D_m : T_{m1}, \ldots, & T_{mn}, \; Z & \rightarrow & P_m \end{array}$$

In the codata type definition above:

- The codata type $C(A_1, \ldots, A_k)$ is polymorphic in type variables $\{A_1, \ldots, A_k\}$. Z is called a state variable. The union of all the type variables used in the different destructors of the codata type is $\{Z, A_1, \ldots, A_k\}$

- Destructor D_1 takes **"a"** number of input terms of types $T_{11}, \ldots, T_{1a}$ respectively.

- T_{ij} is a type expression such that:

$$Type\ Var\ (T_{ij}) \;\; \subseteq \;\; \{Z, A_1, \ldots, A_k\}$$

- The type expression P_i is the output type of destructor D_i such that:

$$Type\ Var\ (P_i) \;\; \subseteq \;\; \{Z, A_1, \ldots, A_k\}$$

- For a destructor D_f, of the codata type C, which takes **"g"** number of inputs:

  - the type of the unfold function corresponding to that constructor is directly given by the type expression corresponding to the destructor in the codata definition which is:

$$T_{f1}, \ldots, T_{fg} \; \rightarrow \; Z$$

  - the type of the destructor D_f is given by:

$$(T_{f1}, \ldots, T_{fg} \; \rightarrow \; Z) \;\; \left[C(A_1, \ldots, A_k)/Z \right]$$

    The above representation means that in the type expression $T_{f1}, \ldots, T_{fg} \; \rightarrow \; Z$, the type variable Z is substituted with $C(A_1, \ldots, A_k)$.

$$\dfrac{\begin{array}{lcll} x_{11} : F_{11}, \ldots, x_{1a} : F_{1a} & \vdash & t_1 : T_1 & \langle E_1 \rangle \\ \vdots & & \vdots & \\ x_{m1} : F_{m1}, \ldots, x_{mn} : F_{mn} & \vdash & t_m : T_m & \langle E_m \rangle \end{array}}{\Gamma \vdash \begin{array}{c}\textsf{rec} \\ \textsf{of}\end{array} \left| \begin{array}{lllll} D_1 & : & x_{11}, \ldots, x_{1a} & \to & t_1 \\ \vdots & & & \vdots & \vdots \\ D_m & : & x_{m1}, \ldots, x_{mn} & \to & t_m \end{array} \right. : T \quad \left\langle \exists \begin{array}{l} T_1, \ldots, T_m, \\ F_{11}, \ldots, F_{1a}, \\ \quad \vdots \\ F_{m1}, \ldots, F_{mn}, \\ A'_1, \ldots, A'_k \end{array} . \begin{array}{l} T = C'(A'_1, \ldots, A'_k) \\ T_1 = P_1\Big[C(A_1, \ldots, A_k)/Z \ , \ A'_1/A_1, \ldots, A'_k/A_k\Big], \\ \qquad \vdots \\ T_m = P_m\Big[C(A_1, \ldots, A_k)/Z \ , \ A'_1/A_1, \ldots, A'_k/A_k\Big], \\ F_{11} = T_{11}\Big[C(A_1, \ldots, A_k)/Z \ , \ A'_1/A_1, \ldots, A'_k/A_k\Big] \\ \qquad \vdots \\ F_{1a} = T_{1a}\Big[C(A_1, \ldots, A_k)/Z \ , \ A'_1/A_1, \ldots, A'_k/A_k\Big] \\ \qquad \vdots \\ F_{m1} = T_{m1}\Big[C(A_1, \ldots, A_k)/Z \ , \ A'_1/A_1, \ldots, A'_k/A_k\Big] \\ \qquad \vdots \\ F_{mn} = T_{mn}\Big[C(A_1, \ldots, A_k)/Z \ , \ A'_1/A_1, \ldots, A'_k/A_k\Big] \\ E_1, E_2, \ldots, E_m \end{array} \right\rangle} \text{ rec}$$

$$\dfrac{\Gamma \vdash t_1 : T_1 \ \ \langle E_1 \rangle \qquad \ldots \qquad \Gamma \vdash t_n : T_n \ \ \langle E_n \rangle}{\Gamma \vdash (t_1, \ldots, t_n) : T \quad \left\langle \exists \, T_1, \ldots, T_n \, . \begin{array}{l} T = (T_1, \ldots, T_n) \\ , E_1, \ldots, E_n \end{array} \right\rangle} \text{ prod}$$

Table 5: Typing Rule for record and product

As in the case of data type definition, the codata type definition is α-renamed with fresh variables to avoid any naming conflicts when elements of codata definitons are used in the type inference rules.

The below representation means that in the type expression T_{ij}, the type variables $Z, A_1, \ldots, A_k$ have been replaced with fresh variables $Z', A'_1, \ldots, A'_k$ respectively.

$$T_{ij} \ [Z'/Z, A'_1/A_1, \ldots, A'_k/A_k]$$

2.3.1 record and product Term

record term forms a record for a codata type. The type of a record of a codata type is that codata type. products can be thought of as special syntax for records. Products allow the programmers to specify tuples in MPL without specifying the corresponding codata types and the projection functions.

First row in Table provides the type inference rules for record term. The noteworthy points in the typing rule of the record term of codata type $C(A_1, \ldots, A_k)$ defined in Section 2.3 are as follows:

- The output type T of a record is the codata type to which the destructors used in the body of record belong. This relationship is represented by the equation:

$$T \ = \ C(A'_1, \ldots, A'_k)$$

  where $A'_1, \ldots, A'_k$ are fresh variables used instead of $A_1, \ldots, A_k$.

- The type T_i of the term t_i, which corresponds to the destructor D_i in the record, is found by replacing the state variable Z with the name of the codata type $C(A'_1, \ldots, A'_k)$ in the type P_i. P_i is the type expression corresponding to the term t_i in the codata definition. This relationship is expressed by the equation:

$$T_i = P_i\Big[C(A_1, \ldots, A_k)/Z \ , \ A'_1/A_1, \ldots, A'_k/A_k\Big]$$

  where $\big[C(A_1, \ldots, A_k)/Z \ , \ A'_1/A_1, \ldots, A'_k/A_k\big]$ represents a list of substitutions to be made inside the type expression P_i.

- The type F_{ij} of x_{ij}, which is the j^{th} argument of the i^{th} destructor of codata type C, is given by the following equation:

$$F_{ij} = T_{ij}\Big[C(A_1, \ldots, A_k)/Z \ , \ A'_1/A_1, \ldots, A'_k/A_k\Big]$$

  where T_{ij} is the type expression corresponding to the x_{ij} in the codata type expression and $\big[C(A_1, \ldots, A_k)/Z \ , \ A'_1/A_1, \ldots, A'_k/A_k\big]$ represents a list of substitutions to be made inside the type expression T_{ij}.

- $E_1, \ldots, E_m$ are respectively the type equations generated for the terms $t_1, \ldots, t_m$ in the context expanded by the variables associated with the respective destructors of the record.

Second row in Table 5 describes the type inference rule for the product term. Product is an inbuilt type in MPL. $(T_1, \ldots, T_n)$ represents a product type which is the type of an n tuple. In the table, $E_1, \ldots, E_n$ represent the sets of type equations associated with terms $t_1, \ldots, t_n$ respectively.

2.3.2 unfold Term

unfold is a constructive operation used to build codata types by unfolding over an initial value. The typing rule for unfold has been described in Table 6. A few noteworthy points about the type inference rule for unfold are as follows:

- The output type T of an unfold term is the codata type to which the destructors used in the body of the unfold term belong. This relationship is represented by the equation:

$$T \ = \ C(A'_1, \ldots, A'_k)$$

  where $A'_1, \ldots, A'_k$ are fresh variables.

- $E_1, \ldots, E_m$ are respectively the type equations generated for the terms $t_1, \ldots, t_m$ in the context expanded by the variables associated with the respective destructors along with the variable s which represents the state.

$$\frac{t : T_0 \ \langle E_0 \rangle \qquad \begin{array}{lcll} x_{11} : F_{11}, \ldots, x_{1a} : F_{1a} \ , \ s : S_1 & \vdash & t_1 : T_1 & \langle E_m \rangle \\ \vdots & \vdots & \vdots & \\ x_{m1} : F_{m1}, \ldots, x_{mn} : F_{mn} \ , \ s : S_m & \vdash & t_m : T_m & \langle E_1 \rangle \end{array}}{\Gamma \vdash \begin{array}{l} \mathsf{unfold} \\ \mathsf{s =>} \left| \begin{array}{llll} C_1 & : \ x_{11}, \ldots, x_{1a} & \to & t_1 \\ \vdots & & \vdots & \vdots \\ C_m & : \ x_{m1}, \ldots, x_{mn} & \to & t_n \end{array} \right. \\ \mathsf{on \ \ t} \end{array} : T} \ \text{unfold}$$

$$\left\langle \exists \begin{array}{l} T_0, T_1, \ldots, T_m, \\ F_{11}, \ldots, F_{1a}, \\ \quad \vdots \\ F_{m1}, \ldots, F_{mn}, \\ A'_1, \ldots, A'_k \end{array} . \begin{array}{l} T = C(A'_1, \ldots, A'_k), \\ S_1 = T_0, \ldots, S_m = T_0, \\ T_1 = P_1 \left[Z'/Z \ , \ A'_1/A_1, \ldots, A'_k/A_k \right], \\ \qquad \vdots \\ T_m = P_m \left[Z'/Z \ , \ A'_1/A_1, \ldots, A'_k/A_k \right], \\ F_{11} = T_{11} \left[Z'/Z \ , \ A'_1/A_1, \ldots, A'_k/A_k \right], \\ \qquad \vdots \\ F_{1a} = T_{1a} \left[Z'/Z \ , \ A'_1/A_1, \ldots, A'_k/A_k \right], \\ \qquad \vdots \\ F_{m1} = T_{m1} \left[Z'/Z \ , \ A'_1/A_1, \ldots, A'_k/A_k \right], \\ \qquad \vdots \\ F_{mn} = T_{mn} \left[Z'/Z \ , \ A'_1/A_1, \ldots, A'_k/A_k \right], \\ E_0, E_1, E_2, \ldots, E_m \end{array} \right\rangle$$

Table 6: Typing Rules for unfold term

- The type of the state variable s should be the same as the type of the value being unfolded.
- E_0 represents the set of the type equations generated for the the term t being unfolded.
- The type T_i of the term t_i corresponding to the destructor D_i is given by:
$$T_i = P_i\Big[Z'/Z\ ,\ A'_1/A_1, \dots, A'_k/A_k\Big]$$
where P_i is the type expression associated with the term t_i corresponding to destructor D_i in the codata definition C and $Z'/Z\ ,\ A'_1/A_1, \dots, A'_k/A_k]$ represents a list of substitutions to be made inside the type expression P_i.
- The type F_{ij} of x_{ij}, which is the j^{th} argument of the i^{th} destructor of codata type C, is given by the following equation:
$$F_{ij} = T_{ij}\Big[Z'/Z\ ,\ A'_1/A_1, \dots, A'_k/A_k\Big]$$
where T_{ij} is the type expression corresponding to x_{ij} in the codata type expression and $\big[Z'/Z\ ,\ A'_1/A_1, \dots, A'_k/A_k\big]$ represents a list of substitutions to be made inside the type expression T_{ij}.

2.3.3 destructor Term

destructor destructs a record of a codata type with a destructor of that codata type to get a value. The value obtained is the the term corresponding to that destructor in the record.

Table 7 describes the typing rule for the destructor term. A few noteworthy points about the type inference rule for a destructor term of codata type C (defined in Section 2.3) are as below:

- destructor takes the following inputs:
  - destructor D_i with its arguments $a_1, \dots, a_j$ and
  - the record of the same codata type of which D_i is a destructor.
- The output type T of a destructor D_i is the same as the type T_0 of the term t_i. t_i is the term corresponding to the branch of destructor D_i in the record passed as the argument to the destructor term. The type T_0 is obtained by the below equation:
$$T_0 = P_i\Big[C(A_1, \dots, A_k)/Z\ ,\ A'_1/A_1, \dots, A'_k/A_k\Big]$$
where P_i is the type expression associated with the term t_i corresponding to destructor D_i in the codata definition C and $\big[Z'/Z\ ,\ A'_1/A_1, \dots, A'_k/A_k\big]$ represents a list of substitutions to be made inside the type expression P_i.
- E_0 is the set of type equations associated with the term t_i in the context expanded by the variables associated with the branch of destructor D_i.
- The type F_j of x_{ij}, which is the j^{th} argument of the i^{th} destructor of codata type C, is given by the following equation:
$$F_j = T_{ij}\Big[Z'/Z\ ,\ A'_1/A_1, \dots, A'_k/A_k\Big]$$
where T_{ij} is the type expression corresponding to x_{ij} in the codata type definition and $\big[Z'/Z\ ,\ A'_1/A_1, \dots, A'_k/A_k\big]$ represents a list of substitutions to be made inside the type expression T_{ij}.

$$
\dfrac{\begin{array}{l}\Gamma \vdash a_1 : T_1 \quad \langle E_{i1} \rangle \\ \qquad\vdots \\ \Gamma \vdash a_j : T_j \quad \langle E_{ij} \rangle\end{array} \qquad \Gamma, x_{i1} : F_1, \ldots, x_{ij} : F_j \vdash t_i : T_0 \quad \langle E_0 \rangle}{\Gamma \vdash \mathsf{dest}\ (D_i, [a_1, \ldots, a_j]) \overset{\mathsf{rec}}{\left(\mathsf{of} \left| \begin{array}{lcl} \vdots & \vdots & \vdots \\ D_i\ x_{i1}, \ldots, x_{ij} & \rightarrow & t_i \\ \vdots & \vdots & \vdots \end{array} \right. \right)} : T} \ \text{dest}
$$

$$
\left\langle \exists \begin{array}{l} T_0, T_1, \ldots, T_j, \\ F_1, \ldots, F_j, \\ A'_1, \ldots, A'_n \end{array} . \begin{array}{l} T = T_0, \\ T_0 = P_i\Big[C(A_1, \ldots, A_k)/Z\ ,\ A'_1/A_1, \ldots, A'_k/A_k\Big], \\ T_1 = S_1, \ldots, T_j = S_j \\ F_1 = T_{i1}\ \Big[C(A_1, \ldots, A_k)/Z\ ,\ A'_1/A_1, \ldots, A'_k/A_k\Big], \\ \qquad \vdots \\ F_j = T_{ij}\ \Big[C(A_1, \ldots, A_k)/Z\ ,\ A'_1/A_1, \ldots, A'_k/A_k\Big], \\ E_0, E_1, \ldots, E_j \end{array} \right\rangle
$$

Table 7: Typing Rules for destructor term

2.4 Other Terms

These are the remaining sequential MPL term. These include variables, constants, function call, if-then-else, switch and where. Sections 2.4.1 through 2.4.5 describe the typing rules for the above mentioned terms.

2.4.1 variables

The type inference rule for variables is provided in the first row of Table 8. The variable being type inferred should be present in the context Γ. In Table 8, the type T of a variable term is equated with the type P of that variable obtained from the context.

2.4.2 constants

The type inference rules for constants are provided in the second row of Table 8. Constants are particularly easy to type infer as different constants are represented with different constructors in the abstract syntax tree of MPL.

2.4.3 function call

Function call is used to call an already defined function with some arguments.

Consider a function f which takes m inputs of types $S_1, \ldots, S_m$. The output type of the function is S. $A_1,\ \ldots,\ A_k$ represent the union of all the type variables in the input and the output types.

$$f : \forall A_1,\ \ldots,\ A_k.\ S_1, \ldots, S_m \rightarrow S$$

Variables
$\dfrac{}{x : P, \Gamma \vdash x : T \quad \left\langle T = P \right\rangle}$ variable
Constants
$\dfrac{\Gamma \vdash n : Int}{\Gamma \vdash \ n : T \quad \left\langle T = Int \right\rangle}$ int $\dfrac{\Gamma \vdash n : Double}{\Gamma \vdash \ n : T \quad \left\langle T = Double \right\rangle}$ double $\dfrac{\Gamma \vdash n : Char}{\Gamma \vdash \ n : T \quad \left\langle T = Char \right\rangle}$ char

Table 8: Typing Rule for variables and constants

$$\frac{\Gamma \vdash t_1 : T_1 \ \ \langle E_1 \rangle \qquad \dots \qquad \Gamma \vdash t_n : T_n \ \ \langle E_n \rangle}{\Gamma \vdash \mathsf{f}\ (\mathsf{t_1}, \dots, \mathsf{t_m}) : T \ \left\langle \exists \begin{array}{l} A'_1, \dots, A'_k, \\ T_1, \dots, T_m \end{array} . \begin{array}{l} T = S\ \left[A'_1/A_1, \dots, A'_k/A_k\right], \\ T_1 = S_1\ \left[A'_1/A_1, \dots, A'_k/A_k\right], \\ \qquad \vdots, \\ T_m = S_m\ \left[A'_1/A_1, \dots, A'_k/A_k\right], \\ E_1, \dots, E_m \end{array} \right\rangle} \text{ call}$$

Table 9: Typing Rule for function call

The type of a function definition is inserted in a data structure called symbol table which is an important data structure used during type inference process. When a function call is made, the type of the function being called can be looked up from the symbol table.

The output and input types are α-renamed before using them in the type equations to avoid variable name clashes between variables used in the input and output types of the function definition obtained from the symbol table and the variables which are already used in the type equation. The α-renamed input and output types are as below:

$$S_i \left[A'_1/A_1, \dots, A'_k/A_k\right] \quad \{\ 1\ \le\ i\ \le m\}$$
$$S \left[A'_1/A_1, \dots, A'_k/A_k\right]$$

In the representation above $A'_1, \dots, A'_k$ are the fresh type variables used instead of $A_1, \dots, A_k$ respectively. $S\ \left[A'_1/A_1, \dots, A'_k/A_k\right]$ represents a list of substitutions such that A'_1 is substituted for A_1, A'_2 for A'_2 and so on inside the body of the type expression S.

Typing rule for function call is provided in Table 9. Important points about the type inference rule for function call in Table 9 are as below:

- The type of a function call is the output type of the function type of the function being called. This relationship is represented by the following equation in the inference rule:

  $$T = S\ \left[A'_1/A_1, \dots, A'_k/A_k\right]$$

  where $\left[A'_1/A_1, \dots, A'_k/A_k\right]$ are the list of substitutions used to α-rename the old variables.

- The type of the various input arguments $T_1, \dots, T_n$ of the function call can be deduced from the input types of the function being called. This relationship is given by the following equations in the inference rule:

  $$T_1 = S_1\ \left[A'_1/A_1, \dots, A'_k/A_k\right],$$
  $$\vdots$$
  $$T_m = S_m\ \left[A'_1/A_1, \dots, A'_k/A_k\right]$$

2.4.4 if-then-else

The type inference rule for an if term is described in the first row of Table 10.

if-then-else term
$\dfrac{\Gamma \vdash t_1 : T_1 \ \langle E_1\rangle \qquad \Gamma \vdash t_2 : T_2 \ \langle E_2\rangle \qquad \Gamma \vdash t_3 : T_3 \ \langle E_3\rangle}{\Gamma \vdash \mathsf{if}\ \mathsf{t}_1\ \mathsf{then}\ \mathsf{t}_2\ \mathsf{else}\ \mathsf{t}_3 : T \ \left\langle \exists\, T_1, T_2, T_3 \ . \begin{array}{l} T_1 = Bool, \\ T_2 = T, T_3 = T, \\ E_1, E_2, E_3 \end{array}\right\rangle}\ \text{if}$
switch term
$\dfrac{\begin{array}{ll} \Gamma \vdash p_1 : P_1 \ \langle E_{p,1}\rangle & \Gamma \vdash t_1 : T_1 \ \langle E_1\rangle \\ \vdots & \vdots \\ \Gamma \vdash p_m : P_m \ \langle E_{p,m}\rangle & \Gamma \vdash t_{m+1} : T_{m+1} \ \langle E_{m+1}\rangle \end{array}}{\Gamma \vdash \begin{array}{l} \mathsf{switch} \\ \left\| \begin{array}{lll} p_1 & = & t_1 \\ \vdots & \vdots & \vdots \\ p_m & = & t_m \\ \texttt{default} & = & t_{m+1} \end{array}\right. \end{array} : T \ \left\langle \exists \begin{array}{l} T_1,\dots,T_{m+1}, \\ P_1,\dots,P_m \ . \end{array} \begin{array}{l} P_1 = Bool, \dots, P_m = Bool \\ T_1 = T, \dots, T_{m+1} = T, \\ E_1,\dots,E_{m+1}, \\ E_{p,1},\dots,E_{p,m} \end{array}\right\rangle}\ \text{switch}$
where term
$\dfrac{\begin{array}{l} \Gamma \vdash t_1 : T_1 \ \langle E_1\rangle \\ \vdots \\ \Gamma \vdash t_m : T_m \ \langle E_m\rangle \end{array} \qquad \Gamma, x_1 : S_1, \dots x_m : S_m \vdash t : T_0 \ \langle E_0\rangle}{\Gamma \vdash \begin{array}{l} t \\ \mathsf{where} \\ \left\| \begin{array}{lll} & \mathsf{fdefn}_1, & \\ & \vdots & \\ & \mathsf{fdefn}_n, & \\ x_1 & = & t_1 \\ \vdots & \vdots & \vdots \\ x_m & = & t_m \end{array}\right. \end{array} : T \ \left\langle \exists \begin{array}{l} T_0,\dots,T_m, \\ S_1,\dots,S_m \ . \end{array} \begin{array}{l} T = T_0, \\ S_1 = T_1, \dots, S_m = T_m, \\ E_0, E_1, \dots, E_m \end{array}\right\rangle}\ \text{where}$

Table 10: Typing Rule for if-then-else, switch and where terms

if term takes three terms as arguments, the first argument evaluates to a boolean value and of the remaining two terms, one is associated with the **then** clause and the other is associated with the **else** clause. Depending on the boolean value being **True** or **False**, first or the second term is executed. The output type of the if statement is the type of the terms associated with the **then** or the **else** clause. A correctly typed if statement will have the two terms of the same type.

2.4.5 switch

A switch term (boolean guard) consists of a list of pair of terms. The first term of each pair evaluates to a boolean value. The first pair for which the first term of the pair evaluates to **True**, the second element of that pair which is again a term is executed.

The type inference rule for switch is provided in the second row of Table 10. A few noteworthy points about the type inference rule for switch provided in Table 10 are as follows:

- The type of the first terms of each pair of the list of m pairs, represented by type variables $P_1, \ldots, P_m$, associated with the switch term is **Bool** and is given by the below equation:

$$P_1 = Bool, \ldots, P_m = Bool$$

- The output type of the switch term given by a type variable T is the type of the second element of a pair from the list of pairs associated with the switch term. In a well typed switch term, the type of the second elements of each pair from the pair list is the same. The equation expressing this relationship is:

$$T_i = T, \ldots, T_m = T$$

  where $T_1, \ldots, T_m$ are respectively the types associated with the second elements of $m + 1$ pairs of the switch term.

- $E_{p,1}, \ldots, E_{p,m}$ represent the set of type equations generated corresponding to the first terms of the list of pairs.

- $E_1, \ldots, E_{m+1}$ represent the set of type equations generated corresponding to the second terms of the list of pairs.

2.4.6 where

where term allows programmers to define local constants and local functions. The third row in Table 10 represents the type inference rules for the where term.

A few noteworthy points about the type inference rule for the where term in Table 10 are as follows:

- The local functions represented by $fdefn_1, \ldots, fdefn_n$ in the type equations are type inferred first. The inferred types of the local functions are then added to the symbol table which can be looked up. The scope of these functions is just the where term.

- The output type T of a where term is the type of the term t which is represented by T_0 in the type inference rule for where.

- E_0 is the set of equations generated by the term t in the context expanded by the local constants $x_1, \ldots, x_m$.

- $E_1, \ldots, E_m$ are the set of equations respectively associated with the terms $t_1, \ldots, t_m$ in the **where** clause of the where term.

$$\frac{\begin{array}{l} \{\ \} \vdash p_1 : T_1 \quad \left\langle \Gamma_1, E_1 \right\rangle \\ \qquad \vdots \\ \{\ \} \vdash p_n : T_n \quad \left\langle \Gamma_n, E_n \right\rangle \end{array} \qquad \Gamma_1 \cup \ldots \cup \Gamma_n \vdash t : T_0 \ \left\langle E \right\rangle}{\{\ \} \vdash (\mathsf{p_1}, \ldots, \mathsf{p_n} \ \rightarrow \ \mathsf{t}\ ) : T \quad \left\langle \exists\, T_0, \ldots, T_n. \begin{array}{l} T = T_1, \ldots, T_n \ \rightarrow \ T_0, \\ E, E_1, \ldots, E_n \end{array} \right\rangle} \text{pattPhr}$$

Table 11: Typing Rule for pattern phrase

3 Generating Type Equations for a Pattern Phrase

A function body is made up of a list of pattern phrases. Recall that a pattern phrase is a list of patterns and the corresponding term in a function definition. This section deals with the generation of type equations for a pattern phrase.

Table 11 describes the type inference rule for a pattern phrase. Noteworthy points about the type inference rules provided in Table 11 are as follows:

- The type equation generation for the pattern phrase $(\mathsf{p_1}, \ldots, \mathsf{p_n} \ \rightarrow \ \mathsf{t})$ is started with an empty context reprsented by { }.
- Type inference process of a pattern enhances the context that it takes as input in addition to generating the type equations. In this regard, patterns are different from terms which only generate type equations using the given context. Thus type inference for pattern p_i results in a new context Γ_i and a set of equations E_i represented as $\left\langle \Gamma_i, E_i \right\rangle$.
- The term t of the pattern phrase is type inferred in the final context generated by the union of the contexts obtained from each pattern in the pattern phrase. For pattern $p_1, \ldots, p_n$, the final context in which the term t is type inferred is represented as $\Gamma_1 \ \cup \ldots \cup \ \Gamma_n$.
- E is the set of equations generated for term t.
- if $T_1, \ldots, T_n$ are the types corresponding to the patterns $p_1, \ldots, p_n$ respectively and T_0 is the type of the term t, then the type of the pattern phrase represented by T is given by the equation:

$$T = T_1, \ldots, T_n \ \rightarrow \ T_0$$

  Symbol $\rightarrow$ in the type expression above shows that T is a function type.

The type equation generation scheme for a pattern phrase discussed above requires the types of the patterns in the pattern phrase. The type inference rules for various patterns have been discussed in Section 3.1.

3.1 Generating Type Equations for Patterns

MPL provides following kinds of patterns:

- Don't Care Pattern

Don't Care Pattern
$\dfrac{}{\{\ \} \vdash \_ : T \quad \left\langle \{\}, \{\} \right\rangle}\ \text{Don}'\text{t Care. patt}$
Variable Pattern
$\dfrac{}{\{\ \} \vdash \mathsf{VPatt}\ \ \mathsf{x} : T \quad \left\langle \{x : T\}, \{\} \right\rangle}\ \text{var. patt}$
Constructor Pattern
$\dfrac{\begin{array}{c} \{\ \} \vdash p_1 : T_1 \qquad \left\langle \Gamma_1, E_1 \right\rangle \\ \vdots \\ \{\ \} \vdash p_j : T_j \qquad \left\langle \Gamma_j, E_j \right\rangle \end{array}}{\{\ \}\ \vdash \mathsf{CPatt}\ \ (\mathsf{C_i}, [\mathsf{p_1}, \ldots, \mathsf{p_j}]) : T}\ \text{cons. patt}$ $\left\langle\ \Gamma_1, \ldots, \Gamma_m\ , \left\{ \exists\ \begin{array}{l} A_1, \ldots, A_k \\ T_1, \ldots, T_j \end{array}\ .\ \begin{array}{l} T = D(A'_1, \ldots, A'_k), \\ T_1 = T_{i1}\Big[D(A_1, \ldots, A_k/Z)\ ,\ A'_1/A_1, A'_k/A_k\Big], \\ \qquad\vdots \\ T_j = T_{ij}\Big[D(A_1, \ldots, A_k/Z)\ ,\ A'_1/A_1, A'_k/A_k\Big], \\ E_1, \ldots, E_j \end{array} \right\}\ \right\rangle$

Table 12: Typing Rule for don't care, variable, constructor patterns

Record Pattern
$$\frac{\begin{array}{c}\{\ \} \vdash p_1 : T_1 \quad \left\langle \Gamma_1, E_1 \right\rangle \\ \vdots \\ \{\ \} \vdash p_m : T_m \quad \left\langle \Gamma_m, E_m \right\rangle\end{array}}{\{\ \} \vdash \overset{\mathsf{RPatt}}{\mathsf{of}} \left\vert \begin{array}{ccc} D_1 & : & p_1 \\ \vdots & & \vdots \\ D_m & : & p_m \end{array}\right. : T \quad \left\langle \Gamma_1, \ldots, \Gamma_m , \left\{ \exists \begin{array}{l} A_1, \ldots, A_k, \\ T_1, \ldots, T_m \end{array} . \begin{array}{l} T = C(A'_1, \ldots, A'_k), \\ T_1 = P_1 \left[C(A_1, \ldots, A_k)/Z \ , \ A'_1/A_1, \ldots, A'_1/A_k \right], \\ \quad \vdots \\ T_m = P_m \left[C(A_1, \ldots, A_k)/Z \ , \ A'_1/A_1, \ldots, A'_1/A_k \right], \\ E_1, \ldots, E_m \end{array} \right\} \right\rangle} \text{ rec. patt}$$
Product Pattern
$$\frac{\begin{array}{c}\{\ \} \vdash p_1 : T_1 \quad \left\langle \Gamma_1, E_1 \right\rangle \\ \vdots \\ \{\ \} \vdash p_j : T_j \quad \left\langle \Gamma_j, E_j \right\rangle\end{array}}{\{\ \} \vdash \mathsf{PPatt}\ (\mathsf{p_1}, \ldots, \mathsf{p_j}) : T} \text{ prod. patt}$$ $$\left\langle \Gamma_1, \ldots, \Gamma_m , \left\{ \exists \begin{array}{l} A_1, \ldots, A_k, \\ T_1, \ldots, T_j \end{array} . \begin{array}{l} T = (T_1, \ldots, T_j), \\ E_1, \ldots, E_j \end{array} \right\} \right\rangle$$

Table 13: Typing Rule for record and product patterns

- Variable Pattern
- Constructor Pattern
- Record Pattern
- Product Pattern

In the following sections, the type inference rules for various patterns are discussed. The type inference process for the various patterns yields a context in addition to type equations. The contexts yielded by the various patterns of a pattern phrase are then unioned to come up with the context in which the corresponding term of that pattern phrase is type inferred. The type inference rules for various patterns have been listed in Table 12 and 13.

3.1.1 Don't Care Patterns

Don't care pattern is used in a pattern phrase when a particular pattern is not relevant for the corresponding term of that pattern phrase. The first row of 12 provides the typing rule for the don't care pattern.

As row one of Table 12 shows, the don't care patterns don't contribute anything to either the context or to the set of type equations.

3.1.2 Variable Patterns

In a pattern phrase, the variables introduced by the variable patterns can be used inside the body of the term corresponding to that pattern phrase.

As can be seen in the second row of Table 12, a variable pattern extends the context by adding the variable of the pattern with a type variable in the context. However, a variable pattern doesn't add anything to the set of equations.

3.1.3 Constructor Patterns

Constructor patterns can be used to pattern match on the constructors of a data type. The third row of Table 12 describe the typing rule for a constructor pattern.

For the purposes of this discussion assume that the constructors used in the constructor pattern belong to the data type $D(A_1,\ldots,A_k)$ defined in Section 2.1.1. $D(A_1,\ldots,A_k)$ has m constructors from $C_1,\ldots,C_m$. In the constructor pattern in the third of Table 12, constructor C_i is used which takes j number of arguments.

A few noteworthy points about the type inference rule for constructor pattern of constructor C_i provided in Table 2 are as follows:

- $\Gamma_1,\ldots,\Gamma_j$ are the contexts associated with the arguments $p_1,\ldots,p_j$ of C_i which are patterns themselves.
- The type equations generated for the constructor pattern formed from constructor C_i are the same as the type equations generated for the constructor term made up of constructor C_i.

- The output type of a constructor pattern term is the data type of which it is a constructor. The equation corresponding to this relationship is:

$$T \ = \ D(A'_1, \ldots, A'_k)$$

where $A'_1, \ldots, A'_k$ are fresh variables used instead of the variables $A_1, \ldots, A_k$.

- The types of the various arguments of the constructor are the type expressions at the corresponding positions of that data constructor in the data definition with the state variable Z substituted with $D(A'_1, \ldots, A'_k)$. The type T_j of the $\mathbf{j^{th}}$ argument of constructor C_i is given by:

$$T_j = T_{ij} \ \big[D(A_1, \ldots, A_k)/Z \ , \ A'_1/A_1, \ldots, A'_k/A_k\big]$$

where T_{ij} is the type expression corresponding to the j^{th} argument of the constructor C_i in the data definition of D. Syntax $T_{ij} \ \big[D(A_1, \ldots, A_k)/Z \ , \ A'_1/A_1, \ldots, A'_k/A_k\big]$ represents that in the body of type expression T_{ij}, a list of substitutions $\big[D(A_1, \ldots, A_k)/Z \ , \ A'_1/A_1, \ldots, A'_k/A_k\big]$ are used.

- E_1, …, E_j are the type equations generated for the terms t_1, …, t_j respectively.

3.1.4 Record Pattern

One can pattern match on records of a first order codata type in MPL.

Consider a first order codata type C defined below:

$$\begin{array}{llll} \mathit{codata}\ Z \ \rightarrow \ C(A_1, \ldots, A_k) = & D_1 : \ Z & \rightarrow & P_1 \\ & \vdots & \vdots & \vdots \\ & D_m : \ Z & \rightarrow & P_m \end{array}$$

In the codata type defintion above:

- $D_1, \ldots, D_m$ are the destructors of the codata type C.
- The codata type is polymorphic in type variables $A_1, \ldots, A_k$.
- The type of a destructor D_i is obtained by using the substitution $\big[C(A_1, \ldots, A_k)/Z\big]$ in the type expression $Z \ \rightarrow \ P_i$ represented as follows:

$$(Z \ \rightarrow \ P_i) \ \big[C(A_1, \ldots, A_k)/Z\big]$$

- P_i is a type expression such that:

$$\mathit{Type\ Var}(P_i) \ \subseteq \{Z, A_1, \ldots, A_k\}$$

- The following representation means that the type variables $A_1, \ldots, A_k$ of the type expression P_i is renamed to fresh variable names $A'_1, \ldots, A'_k$:

$$P_i \ [A'_1/A_1, \ldots, A'_k/A_k]$$

The type inference rule for record patterns is provided in the first row of Table 13. A few noteworthy point about the type inference rule for record pattern provided in Table 13 are as follows:

- $\Gamma_1, \ldots, \Gamma_m$ are the contexts associated with with the elements $p_1, \ldots, p_m$ respectively. $p_1, \ldots, p_m$ are the patterns associated with the destructors $D_1, \ldots, D_m$ of the codata type C respectively.
- The output type T of a record pattern is the codata type of the which the given pattern is a record. This relationship is represented by the following equation:

$$T \;=\; C(A'_1, \ldots, A'_k)$$

  where $A'_1, \ldots, A'_k$ are fresh variables used in place of $A_1, \ldots, A_k$ respectively.
- Type T_i of the pattern p_i corresponding to the destructor D_i in the record is given by the following equation:

$$T_i \;=\; P_i\Big[C(A_1, \ldots, A_k)/Z \;,\; A'_1/A_1, \ldots, A'_k/A_k\Big]$$

  In the equation above $\Big[C(A_1, \ldots, A_k)/Z \;,\; A'_1/A_1, \ldots, A'_k/A_k\Big]$ represents the list of substitutions to be made in the body of the type expression P_i.
- $E_1, \ldots, E_j$ are the type equations generated for the patterns $p_1, \ldots, p_j$ respectively.

3.1.5 Product Pattern

Product patterns consist of a tuple of patterns. The type inference rule for product patterns is specified in the second row of Table 13.

A few noteworthy points about the type inference rule for product patterns provided in Table 13 are as follows:

- $\Gamma_1, \ldots, \Gamma_j$ are the contexts associated with with the elements $p_1, \ldots, p_j$ of the product pattern respectively.
- The output type of a product pattern is the product of the types of the invidual elements of the product pattern. This relationship is represented by the equation:

$$T = (T_1, \ldots, T_n)$$

  where ( ) represents the product type which is a built in type in MPL.
- $E_1, \ldots, E_j$ are the type equations generated for the elements $p_1, \ldots, p_j$ of the product pattern respectively.

4 Generating Type Equations for Function Definitions

Function definitions consist of a list of pattern phrases. Type inference rule for a pattern phrase has already been discussed in Section 3. In Section 4.1, the type inference rule for function defintions which consist of a list of pattern phrases is dicussed. MPL allows programmers to annotate a function with its expected type. Section 4.2 deals with how to use the annotated type for the type checking of the functions. Section 4.3 describes the typing rule for mutually recursive function definitions.

Function without an annotated type
$$\dfrac{\begin{array}{c}\{\,\} \vdash (p_{1,1},\ldots,p_{1,n} \ \rightarrow\ t_1\ ) : T_1 \quad \Big\langle E_1 \Big\rangle \\ \vdots \\ \{\,\} \vdash (p_{m,1},\ldots,p_{m,n} \ \rightarrow\ t_m\ ) : T_m \quad \Big\langle E_m \Big\rangle\end{array}}{\begin{array}{l}\{\,\} \vdash \mathsf{fun\ f} \ = \\ \qquad \begin{array}{c}(p_{1,1},\ldots,p_{1,n} \ \rightarrow\ t_1\ ) \\ \vdots \\ (p_{m,1},\ldots,p_{m,n} \ \rightarrow\ t_m\ )\end{array} \ :\ T \quad \left\langle \exists\, T_1,\ldots,T_m. \begin{array}{l} T_1 = T,\ldots,T_m = T, \\ E_1,\ldots,E_m \end{array} \right\rangle\end{array}}\ \text{fun}$$
Function with an annotated type
$$\dfrac{\begin{array}{l}\{\,\} \vdash \mathsf{fun\ f} \ = \\ \qquad \begin{array}{c}(p_{1,1},\ldots,p_{1,n} \ \rightarrow\ t_1\ ) \\ \vdots \\ (p_{m,1},\ldots,p_{m,n} \ \rightarrow\ t_m\ )\end{array} \ :\ T \quad \Big\langle E_{fun} \Big\rangle\end{array}}{\begin{array}{l}\{\,\} \vdash \mathsf{fun\ f} \ :\ \forall\, A_1,\ldots,A_k\ .\ F \ = \\ \qquad \begin{array}{c}(p_{1,1},\ldots,p_{1,n} \ \rightarrow\ t_1\ ) \\ \vdots \\ (p_{m,1},\ldots,p_{m,n} \ \rightarrow\ t_m\ )\end{array} \ :\ T \\ \qquad\qquad \left\langle \begin{array}{ll} \forall\, A'_1,\ldots,A'_k, & T \ = \ F\ [A'_1/A_1,\ldots,A'_k/A_k], \\ \exists\, T \qquad . & E_{fun} \end{array} \right\rangle\end{array}}\ \text{annot. fun}$$

Table 14: Typing Rule for Function Definitions

4.1 Function Definitions Without an Annotated Type

First row of Table 14 provides the type inference rules for functions that don't have an annotated type. Noteworthy points about the type inference rule for function definitons without an annotated type provided in Table 14 are as follows:

- The types of different pattern phrases should be the same. This type is also the type of the function f. This relationship is expressed by the below equation:

  $$T_1 = T, \ldots, T_m = T$$

  where $T_1, \ldots, T_m$ are the types of the m pattern phrases and T is the type of the function f.

- $E_1, \ldots, E_m$ represent the type equations respectively associated with the m pattern phrases.

4.2 Function Definitions With an Annotated Type

Second row of Table 14 describes type inference rule for a function that has been annotated with a type by the programmer. Description of the type inference rule for type annotated function definitions provided in Table 14 is as below:

- The annotated type for function f is shown as $\forall\ A_1, \ldots, A_k\ .\ F$ in the type inference rule. Here F is the type expression representing the type of the function f and $\forall\ A_1, \ldots, A_k$ represents the type variables in the type expression F.

- The strategy used to type check function f is: Infer the type of function f getting rid of its annotated type and try to unify the inferred type with the annotated type. If the unification is possible then the annotated type of the function is right otherwise there is a discrepancy between the two types which should result in a type error. Note that type inference yields the most general type of a function whereas the annotated type may be a specific instance of the most general type of the function.

- The equation $T\ =\ F\ [A'_1/A_1, \ldots, A'_k/A_k]$ in the second row of Table 14 represents that substitutions $[A'_1/A_1, \ldots, A'_k/A_k]$ are made inside the body of the type expression F. This is done in order to refresh the variable names of the type variables used in F.

- E_{fun} is the set of functions generated for the function f without its annotated type.

4.3 Mutually Recursive Function Definitions

Two functions **f** and **g** are mutually recursive if the function **f** calls the function **g** and the function **g** calls the function **f**. Similarly, a set of functions are mutually recursive if there is a circular dependency between them.

Table 15 provides the type inference rule for mutually recursive functions. The basic idea of the inference rule is that the set of mutually recursive functions are considered one function for the purpose of the type inference process: the type equations for mutually recursive processes are generated and solved together.

The description of the type inference rule for a set of mutually recursive functions $f_1, \ldots, f_k$ provided in Table 15 is given below:

Mutually recursive functions

$$
\begin{array}{c}
\begin{array}{ccc}
\{\ \} \vdash (p_{1,1}, \ldots, p_{1,n} \;\to\; t_1\ ) : T_1 \quad \Big\langle F_1 \Big\rangle & & \{\ \} \vdash (q_{1,1}, \ldots, q_{1,y} \;\to\; t_1\ ) : S_1 \quad \Big\langle G_1 \Big\rangle \\
\vdots & & \vdots \\
\{\ \} \vdash (p_{m,1}, \ldots, p_{m,n} \;\to\; t_m\ ) : T_m \quad \Big\langle F_m \Big\rangle & \ldots & \{\ \} \vdash (q_{x,1}, \ldots, q_{x,y} \;\to\; t_x\ ) : S_x \quad \Big\langle G_x \Big\rangle
\end{array} \\
\hline
\begin{array}{l}
\{\ \} \;\vdash \mathsf{mutual} \\
\qquad \mathsf{fun}\ \ \mathsf{f_1} \;= \\
\qquad\quad \left.\begin{array}{l}
(\mathsf{p_{1,1}}, \ldots, \mathsf{p_{1,n}} \;\to\; \mathsf{t_1}\ ) \\
\qquad\quad \vdots \\
(\mathsf{p_{m,1}}, \ldots, \mathsf{p_{m,n}} \;\to\; \mathsf{t_m}\ )
\end{array}\right. \quad : \; T \\
\qquad\qquad \vdots \\
\qquad \mathsf{fun}\ \ \mathsf{f_k} \;= \\
\qquad\quad \left.\begin{array}{l}
(\mathsf{q_{1,1}}, \ldots, \mathsf{q_{1,y}} \;\to\; \mathsf{t_1}\ ) \\
\qquad\quad \vdots \\
(\mathsf{q_{x,1}}, \ldots, \mathsf{p_{x,y}} \;\to\; \mathsf{t_x}\ )
\end{array}\right. \quad : \; S \quad
\left\langle \exists \begin{array}{c} T_1, \ldots, T_m \\ \vdots \\ S_1, \ldots, S_x\ . \end{array}
\begin{array}{l} T_1 = T, \ldots, T_m = T, \\ \qquad \vdots \\ S_1 = S, \ldots, S_m = S, \\ F_1, \ldots, F_m, \\ \quad \vdots \\ G_1, \ldots, G_x \end{array} \right\rangle
\end{array}
\end{array}
$$

Table 15: Typing Rule for Mutually Recursive Function Definitions

- The functions $f_1, \ldots, f_k$ are assigned dummy types which are then inserted in the symbol table. In this way the problem of calling a function before it is defined, as is the case in mutually recursive functions, is addressed.
- The equations for individual functions are generated and combined to get the final equation for the set of mutually recursive functions.