Typing Rules for Concurrent MPL

March 14, 2017

1 Type Inferencing

1.1 Introduction

This chapter deals with the type inferencing of various MPL constructs, namely functions, processes, terms and process commands when they are not type annotated and type checking when they are. Type inference process involves two main steps.

- Generating Type Equations - Type Equations represent the constraint relationship between the different parts of the construct being type inferred. $\exists$ used in the typing rules here is not be existential quantification from the predicate logic rather it is used a notation as a type binder.

  Using the typing rules, a series of equations are generated going from botton to top or the proof search direction. The equation structure is not linear rather it is hierarchical capturing the shape of the type inference tree. Thus,placement of an equation inside a type equation roughly resembles the location within a type inference tree that resulted in that equation within the type equation. Advantage of this approach is faster solving of type equations and better localisation of type errors.

- Solving Type Equations - The type equations hence forth generated are solved in order to get the most general type of a given construct.

In this chapter, the type equations generation for the Sequential MPL (terms and functions) is dicussed first followed by that of the Concurrent MPL (process commands and processes). This is followed by an alogrithm to solve the type equations to get the most general type.

2 Generating Type Equations for sequential terms

2.1 var term

In order to infer a variable a variable, it should be looked up in the context. If it is present in the context then both the types of the variables are equated.

Typing rule for var

$$\frac{}{x : P, \Gamma \vdash x : Q \quad \left\langle P = Q \right\rangle}\ \text{id}$$

2.2 default term

default term is used as the catch all branch in the guard branches of the switch statement in the language. The type of the default term is Bool.

Typing rule for default

$$\frac{}{\Gamma \vdash default : T \quad \left\langle T = Bool \right\rangle}\ \text{default}$$

2.3 constant term

constants can be one of the base types, like Int, Double and Char

Typing rule for constants

$$\frac{}{\Gamma \vdash cInt\ n : T \quad \left\langle T = Int \right\rangle}\ \text{int}$$

$$\frac{}{\Gamma \vdash cDouble\ n : T \quad \left\langle T = Double \right\rangle}\ \text{double}$$

$$\frac{}{\Gamma \vdash cChar\ n : T \quad \left\langle T = Char \right\rangle}\ \text{char}$$

2.4 case and cons term

cons term creates a data with the constructors of that data type.

case term branches on the different constructors of the data type. Depending on the branch selected, some code can be executed. The type of a case term is the type of the term to be run when a branch is selected. A well typed case construct will have all the code terms corresponding to all the branches/construcors of the same type.

Let us define a data type D. A_1, ..., A_k is the union of type variables used in the different constructors of the data type. C_1 ... C_m are the different constructors of the data type D. For C_1 ,F_{11}, ..., F_{1a} repesent the input type of the destructor.

The renamed constructs have been represented by adding a subscript N to the name of the construct. The renaming is done to avoid any conflict between the variable names used in the the data construct with that used in the equation.

$$\begin{aligned} \forall A_1,\ \ldots,\ A_k.\ data\ D\ \rightarrow\ A = \ & C_1 : F_{11},\ \ldots,\ F_{1a} \rightarrow A \\ & \vdots \qquad\quad \vdots \qquad\quad \vdots \\ & C_m : F_{m1},\ \ldots,\ F_{mn} \rightarrow A \end{aligned}$$

Renaming the different constructs in the data D with fresh variables A'_1, ..., A'_k gives the following constructs.

$$\begin{aligned} D_N \quad &= (\Lambda A_1, \ldots, A_k.\ D)\ A'_1, \ldots, A'_k \\ F_{xy,N} &= (\Lambda A_1, \ldots, A_k.\ F_{xy})\ A'_1, \ldots, A'_k \quad \{1 \leq x \leq m,\ 1 \leq y \leq p \mid p \in \{a, b, \ldots, n\}\} \end{aligned}$$

The output type of the cons term is the data type of which it is a constructor of. The function type of the constructor is looked up from the symbol table and the different arguments of the constructor are assigned their corresponding types from the input types of the afore mentioned function type.E_1,...,E_n are the type equations generated for the terms t_1, ..., t_n respectively.

The type of the case term is the type of the term corresponding to a branch/constructor in the case construct. A well typed case term will have the same type of the terms corresponding to different branches. The types of the input arguments for a constructor is obtained by looking up the function type for that constructor from the symbol table.

Typing Rule for cons and case

$$\frac{\Gamma \vdash t_1 : T_1 \quad \langle E_1 \rangle \qquad \ldots \qquad \Gamma \vdash t_j : T_j \quad \langle E_j \rangle}{\Gamma \vdash \mathsf{cons}\ (C_i, [t_1, \ldots, t_j]) : T \quad \left\langle \exists \begin{array}{l} T_1, \ldots, T_j, \\ A_1, \ldots, A_n. \end{array} \begin{array}{l} T = D_N \\ T_1 = F_{i1,N},\ \ldots,\ T_j = I_{ij,N} \\ E_1, E_2, \ldots, E_j \end{array} \right\rangle}\ \text{cons}$$

$$\frac{t : T_0\ \langle E_0 \rangle \qquad x_{11} : T_{11}, \ldots, x_{1a} : T_{1a} \vdash t_1 : T_1\ \langle E_1 \rangle \qquad \ldots \qquad x_{m1} : T_{m1}, \ldots, x_{mn} : T_{mn} \vdash t_m : T_m\ \langle E_m \rangle}{\Gamma \vdash \begin{array}{l} \mathsf{case}\ t \\ \mathsf{of} \end{array} \left| \begin{array}{lcl} C_1\ x_{11}, \ldots, x_{1a} & \rightarrow & t_1 \\ \vdots & \vdots & \vdots \\ C_m\ x_{m1}, \ldots, x_{mn} & \rightarrow & t_n \end{array} \right. : T \quad \left\langle \exists \begin{array}{l} T_0, \\ T_1, \ldots, T_m, \\ T_{11}, \ldots, T_{1a}, \\ \vdots \\ T_{m1}, \ldots, T_{mn}, \\ A'_1, \ldots, A'_k \end{array} . \begin{array}{l} T_0 = D_N \\ T_1 = T, \ldots, T_m = T \\ T_{11} = F_{11,N}, \ldots, T_{11} = F_{1a,N} \\ \vdots \\ T_{m1} = F_{m1,N}, \ldots, T_{mn} = F_{mn,N} \\ E_1, E_2, \ldots, E_m \end{array} \right\rangle}\ \text{case}$$

2.5 fold term

fold is used to implement higher order primitive recursive functions which gurantees termination. For all the constructors of a data type, their fold functions are inserted in the symbol table.

The type of the fold term is the type of the term corresponding to a branch/constructor in the fold construct. A well typed fold term will have the same type of the terms corresponding to different branches. The types of the input arguments for a branch is obtained by looking up the fold function for that constructor from the symbol table.

Typing Rule for fold

$$\frac{t : T_0\ \langle E_0 \rangle \qquad x_{11} : T_{11}, \ldots, x_{1a} : T_{1a} \vdash t_1 : T_1\ \langle E_1 \rangle \qquad \ldots \qquad x_{m1} : T_{m1}, \ldots, x_{mn} : T_{mn} \vdash t_m : T_m\ \langle E_m \rangle}{\Gamma \vdash \begin{array}{l} \mathsf{fold}\ t \\ \mathsf{of} \end{array} \left| \begin{array}{lcl} C_1 : x_{11}, \ldots, x_{1a} & \rightarrow & t_1 \\ \vdots & \vdots & \vdots \\ C_m : x_{m1}, \ldots, x_{mn} & \rightarrow & t_n \end{array} \right. : T \quad \left\langle \exists \begin{array}{l} T_0, T_1, \ldots, T_m, \\ T_{11}, \ldots, T_{1a}, \\ \vdots \\ T_{m1}, \ldots, T_{mn}, \\ A'_1, \ldots, A'_k \end{array} . \begin{array}{l} T_0 = D_{new} \\ T_1 = T, \ldots, T_m = T \\ T_{11} = F_{11,N}, \ldots, T_{11} = F_{1a,N} \\ \vdots \\ T_{m1} = F_{m1,N}, \ldots, T_{mn} = F_{mn,N} \\ E_1, E_2, \ldots, E_m \end{array} \right\rangle}\ \text{fold}$$

2.6 record and dest term

record term forms a record for a codata type. dest destructs a record with a particular destuctor and obtain the term corresponding to that destructor from the record.

Let us defined a codata type C, A_1, ..., A_k is the union of type variables used in its different destructors. D_1, ..., D_m are the different destructors of the codata type C. For D_1, F_{11}, ..., F_{1a} repesent the input type of the destructor.

$$\begin{aligned} \forall A_1,\ \ldots,\ A_k.\ \mathit{codata}\ A\ \rightarrow\ C = \ & D_1 : F_{11},\ \ldots,\ F_{1a},\ A \rightarrow O_1 \\ & \vdots \qquad\quad \vdots \qquad\quad \vdots \\ & D_m : F_{m1},\ \ldots,\ F_{mn},\ A \rightarrow O_n \end{aligned}$$

Renaming the different constructs in the codata C with fresh variables A'_1, ..., A'_k gives the following constructs.

$$\begin{aligned} C_N &= (\Lambda A_1, \ldots, A_k.\ C)\ A'_1, \ldots, A'_k \\ F_{xy,N} &= (\Lambda A_1, \ldots, A_k.\ F_{xy})\ A'_1, \ldots, A'_k \quad \{1 \leq x \leq m, 1 \leq y \leq p \mid p \in \{a, b, \ldots, n\}\} \\ O_{i,N} &= (\Lambda A_1, \ldots, A_k.\ O_i)\ A'_1, \ldots, A'_k \quad \{1 \leq i \leq m\} \end{aligned}$$

The type of a record term for a codata type is that codata type. The types of the different detsructor arguments and the term corresponding to a destructor branch is obtained by looking up the function type corresponding to that destructor from the symbol table. The input type gives the type of the constructor arguments and the output type gives the type of the term corresponding to the destructor branch.

The type of a destructor term is the type of term corresponding to that destructor branch in the given record. This is obtained from the function type (output type of the function) of the destructor which can be looked up from the symbol table. The types of the destructor arguments are obtained from the corresponding input type of the afore mentioned function.

Typing Rule for record

$$\frac{\Gamma \vdash t_{i1} : T_1 \quad \langle E_{i1} \rangle \qquad \ldots \qquad \Gamma \vdash t_{ij} : T_j \quad \langle E_{ij} \rangle \qquad \Gamma, x_{i1} : I_1, \ldots, x_{ij} : I_j \vdash t_i : T_0 \quad \langle E_0 \rangle}{\Gamma \vdash \mathsf{dest}\ (D_i, [t_{i1}, \ldots, t_{ij}])\ \left(\begin{array}{l} \mathsf{rec} \\ \mathsf{of} \end{array} \left| \begin{array}{lcl} \vdots & \vdots & \vdots \\ D_i\ x_{i1}, \ldots, x_{ij} & \rightarrow & t_i \\ \vdots & \vdots & \vdots \end{array} \right. \right) : T \left\langle \exists \begin{array}{l} T_0, T_1, \ldots, T_j, \\ A'_1, \ldots, A'_n \end{array} . \begin{array}{l} T = T_0, \\ T_0 = O_{i,N}, T_1 = I_1, \ldots, T_j = I_j \\ I_1 = F_{i1,N}, \ldots, I_j = F_{ij,N}, \\ E_0, E_1, \ldots, E_j \end{array} \right\rangle} \text{ dest}$$

$$\frac{t : T_0\ \langle E_0 \rangle \qquad x_{11} : T_{11}, \ldots, x_{1a} : T_{1a} \vdash t_1 : T_1\ \langle E_1 \rangle \quad \ldots \quad x_{m1} : T_{m1}, \ldots, x_{mn} : T_{mn} \vdash t_m : T_m\ \langle E_m \rangle}{\Gamma \vdash \begin{array}{l} \mathsf{rec} \\ \mathsf{of} \end{array} \left| \begin{array}{lcl} D_1\ x_{11}, \ldots, x_{1a} & \rightarrow & t_1 \\ \vdots & \vdots & \vdots \\ D_m\ x_{m1}, \ldots, x_{mn} & \rightarrow & t_n \end{array} \right. : T \left\langle \exists \begin{array}{l} T_0, T_1, \ldots, T_m, \\ T_{11}, \ldots, T_{1a}, \\ \vdots \\ T_{m1}, \ldots, T_{mn}, \\ A'_1, \ldots, A'_k \end{array} . \begin{array}{l} T = C_N \\ T_1 = O_{1,N}, \ldots, T_m = O_{m,N} \\ T_{11} = F_{11,N}, \ldots, T_{1a} = F_{1a,N} \\ \vdots \\ T_{m1} = F_{m1,N}, \ldots, T_{mn} = F_{mn,N} \\ E_1, E_2, \ldots, E_m \end{array} \right\rangle} \text{ rec}$$

2.7 prod term

prod term is used to reprsent tuples in MPL.The output type of a tuple is the product of the types of the individual elements.

Typing Rule for prod

$$\frac{\Gamma \vdash x_1 : T_1\ \langle E_1 \rangle \qquad \ldots \qquad \Gamma \vdash x_n : T_n\ \langle E_n \rangle}{\Gamma \vdash (x_1, \ldots, x_n) : T \left\langle \exists\ T_1, \ldots, T_n\ . \begin{array}{l} T = (T_1, \ldots, T_n) \\ , E_1, \ldots, E_n \end{array} \right\rangle} \text{ prod}$$

2.8 function call

Function call is used to call an already defined function in the program. The type of the called function is looked up from the symbol table. In the below representation of a function type, function

f's type is comprised of m input types $I_1, \ldots I_m$ and an output type O.

$$f : \forall A_1, \ \ldots, \ A_k.\ I_1, \ldots, I_m \to O$$

As in the previous term case, the output and input types are α renamed.

$$I_{i,N} = (\Lambda A_1, \ldots, A_k.\ I_i)\ A'_1, \ldots, A'_k$$
$$O_N = (\Lambda A_1, \ldots, A_k.\ O)\ A'_1, \ldots, A'_k$$

The type of a function call is the type of the output type of the function being called. To type infer the function arguments, the arguments are type inferred in the original context Γ and these types are equated with their corresponding input types of the function obatined from the symbol table.

Typing Rule for function call

$$\frac{\Gamma \vdash x_1 : X_1 \ \langle E_1 \rangle \qquad \ldots \qquad \Gamma \vdash x_n : X_n \ \langle E_n \rangle}{\Gamma \vdash \mathsf{f}\ (\mathsf{x_1}, \ldots, \mathsf{x_m}) : T \ \left\langle \exists \begin{array}{l} A'_1, \ldots, A'_k, \\ T_1, \ldots, T_m \end{array} . \begin{array}{l} T = O_N, \\ T_1 = I_{1,N}, \ldots, T_m = I_{m,N}, \\ E_1, \ldots, E_m \end{array} \right\rangle} \text{ call}$$

2.9 if term

if is a term with three arguments, a boolean expression and two statements. Depending on the boolean value, first or the second statement is executed. The type of the if statement is the type of the type of these statements. A correctly typed program will have these two statements of the same type.

Typing Rule for if

$$\frac{\Gamma \vdash t_1 : T_1 \ \langle E_1 \rangle \qquad \Gamma \vdash t_2 : T_2 \ \langle E_2 \rangle \qquad \Gamma \vdash t_3 : T_3 \ \langle E_3 \rangle}{\Gamma \vdash \mathsf{if\ t1\ then\ t2\ else\ t3} : T \quad \left\langle \exists\, T_1, T_2, T_3\ .\ \begin{array}{l} T_1 = Bool, \\ T_2 = T, T_3 = T, \\ E_1, E_2, E_3 \end{array} \right\rangle} \text{ if}$$

2.10 Let term

let statment allows local defintions to be used in a term. let also allows the programmer define local functions. The local functions are type inferred and added to the symbol table. The scope of these functions is just the let term. The representation below assumes that these local functions have already been taken care of and all that remains in the where part are local variable/pattern declarations. The type the let statement is the type of the term **t**.

Typing Rule for let

$$\frac{\Gamma \vdash t_1 : T_1 \;\; \langle E_1 \rangle \quad \ldots \quad \Gamma \vdash t_m : T_m \;\; \langle E_m \rangle \qquad \Gamma, x_1 : I_1, \ldots x_m : I_m \vdash t : T_0 \;\; \langle E_0 \rangle}{\Gamma \vdash \begin{array}{l}\textsf{let } t \\ \textsf{where} \\ \left|\begin{array}{lcl} x_1 & = & t_1 \\ \vdots & \vdots & \vdots \\ x_m & = & t_m \end{array}\right.\end{array} : T \;\; \left\langle \exists \begin{array}{l} T_0, \ldots, T_m, \\ I_1, \ldots, I_m \end{array} . \begin{array}{l} T = T_0, \\ I_1 = T_1, \ldots, I_m = T_m, \\ E_0, E_1, \ldots, E_m \end{array} \right\rangle} \text{ let}$$

3 Generating Type Equations for Function Defintions

3.1 Function definitions without an annotated type

These are the function definitions for which the programmer has not annotated the expected type of the function. Once the function defintion is type inferred, the function name is inserted in the symbol table with this type.

The patterns on the left hand side of the function defintion can either be a variable or a constructor patterns. For the ease of describing the rule, lets divide the pattern into two cases.

- **Function with variables as the input argument** (fun defn_1) - To type infer function with variable arguments, the variables of the arguments are added to the context and in this enhanced context the term on the right hand side is type inferred.The input type of the function is the type of the input arguments and the output type is the type of the term.
- **Function with constructor patterns as the input argument** (fun defn_2) - Although a function can have constructor patterns of more than one data type as input, we will limit ourselves to just one for the sake of succintness. The given scheme can easily be extrapolated for the multiple case. The input type of the function is the data type of the constructors on the left hand side and the output type of the function is the type of the terms on the right hand side of the constructor.

Typing rule for function defns with variable patterns

$$\frac{\Gamma, x_1 : I_1,\ \ldots,\ x_m : I_m \vdash t : O \ \langle E \rangle}{\Gamma \vdash \begin{array}{l}\mathsf{fun\ f} \\ \quad \mid\ x_1, \ldots, x_m \ \rightarrow\ t\end{array} : T \ \left\langle \exists\ I_1, \ldots, I_m\ .\ \begin{array}{l} T = (I_1, \ldots, I_m)\ \rightarrow\ O, \\ E \end{array} \right\rangle} \ \text{fdefn}_1$$

Suppose the patterns on the left hand side in the function defintion are constructors of a data type D. A_1, ..., A_k is the union of type variables used in the different constructors of the data type. $C_1 \ldots C_m$ are the different constructors of the data type D.

$$\begin{array}{rl} \forall A_1,\ \ldots,\ A_k.\ data\ D\ \rightarrow\ A = & C_1 : F_{11},\ \ldots,\ F_{1a} \rightarrow A \\ & \vdots \qquad\quad \vdots \qquad\quad \vdots \\ & C_m : F_{m1},\ \ldots,\ F_{mn} \rightarrow A \end{array}$$

Renaming the different constructs in the data D with fresh variables A'_1, ..., A'_k gives the following constructs.

$$\begin{array}{l} D_N \quad = (\Lambda A_1, \ldots, A_k.\ D)\ A'_1, \ldots, A'_k \\ F_{xy,N} = (\Lambda A_1, \ldots, A_k.\ F_{xy})\ A'_1, \ldots, A'_k \quad \{1 \le x \le m,\ 1 \le y \le p \mid p \in \{a, b, \ldots, n\}\} \end{array}$$

Typing rule for function defns with constructor patterns

$$\frac{\Gamma, x_{11} : I_{11},\ \ldots,\ x_{1a} : I_{1a} \vdash t_1 : T_1\ \langle E_1 \rangle \quad \ldots \quad \Gamma, x_{m1} : I_{11},\ \ldots,\ x_{mn} : I_{mn} \vdash t_m : T_m\ \langle E_m \rangle}{\Gamma \vdash \begin{array}{l}\mathsf{fun\ f} \\ \quad \mid\ C_1\ x_{11}, \ldots, x_{1a} \ \rightarrow\ t_1 \\ \quad \mid\ \vdots \qquad\quad \vdots\ \ \vdots \\ \quad \mid\ C_m\ x_{m1}, \ldots, x_{mn} \ \rightarrow\ t_m\end{array} : T \ \left\langle \exists \begin{array}{l} I_{11}, \ldots, I_{1a} \\ \vdots \\ I_{m1}, \ldots, I_{mn} \\ T_1, \ldots, T_m, O \end{array} . \begin{array}{l} T = D_N\ \rightarrow\ O, \\ T_1 = O, \ldots, T_m = O, \\ I_{11} = F_{11,N}, \ldots, I_{1a} = F_{1a,N}, \\ \vdots \\ I_{m1} = F_{m1,N}, \ldots, I_{mn} = F_{mn,N}, \\ E_1, \ldots, E_m \end{array} \right\rangle} \ \text{fdefn}_2$$

3.2 Function defintions with an annotated type

These are the function definitions for which the programmer has annotated the expected type of the function. Once the function is type inferred, an attempt is made to unify the annotated function

type with the inferred one. If the two types can be successfully unified, then the inferred type is line with the expectation of the programmer and the annotated type is inserted in the symbol defintion with the name of the function.

The type equations generated for the annotated functions just differ slightly from the unannotated ones. In the interest of succinctness, we will describe the equation generation only for the variable patterns case. Equation generation for the constructor pattern case can be easily deduced from this.

Typing rule for function defns with variable patterns

$$\frac{\Gamma, x_1 : I_1,\ \ldots,\ x_m : I_m \vdash t : O \ \ \langle E \rangle}{\Gamma \vdash \begin{array}{l}\mathsf{fun\ f} \\ \quad \mid \ x_1, \ldots, x_m \ \ \rightarrow \ \ t \end{array} \ : T \ \ \left\langle \exists\ I_1, \ldots, I_m \ . \ \begin{array}{l} T = (I_1, \ldots, I_m) \ \rightarrow \ O, \\ E \end{array} \right\rangle} \ \text{fdefn}_1$$