Typing Rules for Concurrent MPL

March 16, 2017

1 Type Inferencing

This chapter deals with the generation of the type equations for the Concurrent MPL. The types in the concurrent world are known as protocols, i.e the type of a channel is also called a protocol.

The protocols are formed inductive using the base protocol lsited below.

- Get
- Put
- $\otimes$ (Tensor)
- $\oplus$ (Par)
- Neg
- $\perp$ (Bottom)

In the following sections we will see the typing rules for the various Concurrent MPL Constructs.

1.1 get and put Command

get command, get x α gets a value on a channel α and binds that value to variable x. In order to type infer get, the variable x should be added to the sequential context Φ. The polarity of the channel decides whether the channel has a **Get** or a **Put** protocol. This is necessary to ensure that the protocols at the two ends of a channel connecting two processes match. Type of the variable is the first argument to the **Get/Put** protocol. The second argument of the **Get/Put** protocol is obtained by type inferring the channel α from the remaining process commands in the enhanced sequential context.

put command, put t α puts a seqential term t on the channel α. Like with get, polarity of α decides whether it ends up with a **Get** or a **Put** protocol. The term t is type inferred in the sequential context Φ to get the first argument of the **Get/Put** protocol. The second argument of the **Get/Put** protocol is obtained by type inferring the channel α from the remaining process commands in the original sequential context, Φ.

Typing Rules for Get-Put

$$\frac{s :: x : X, \Phi \mid \Gamma, \alpha : S \Vdash \Delta \quad \langle E \rangle}{\mathsf{get}\ \mathsf{x}\ \alpha\ .s :: \Phi \mid \Gamma, \alpha : T \Vdash \Delta \quad \left\langle \exists X, S. T = Put(X,S), E \right\rangle} \text{get}$$

$$\frac{s :: x : X, \Phi \mid \Gamma \Vdash \alpha : S, \Delta \quad \langle E \rangle}{\mathsf{get}\ \mathsf{x}\ \alpha\ .s :: \Phi \mid \Gamma \Vdash \alpha : T, \Delta \quad \left\langle \exists X, S. T = Get(X,S), E \right\rangle} \text{get}$$

$$\frac{\Phi \vdash t : X \quad \langle E_1 \rangle \quad s :: \Phi \mid \Gamma, \alpha : S \Vdash \Delta \quad \langle E_2 \rangle}{\mathsf{put}\ \mathsf{t}\ \alpha\ .s :: \Phi \mid \Gamma, \alpha : T \Vdash \Delta \quad \left\langle \exists X, S. T = Get(X,S), E_1, E_2 \right\rangle} \text{put}$$

$$\frac{\Phi \vdash t : X \quad \langle E_1 \rangle \quad s :: \Phi \mid \Gamma \Vdash \alpha : S, \Delta \quad \langle E_2 \rangle}{\mathsf{put}\ \mathsf{t}\ \alpha\ .s :: \Phi \mid \Gamma \Vdash \alpha : T, \Delta \quad \left\langle \exists X, S. T = Put(X,S), E_1, E_2 \right\rangle} \text{put}$$

1.2 Close and Halt Command

The protocol as a result of close/halt is $\perp$. When a channel is the last remaining channel in the input or the output context, then only it can be halted.

Typing Rules for close-halt

$$\frac{s :: \Phi \mid \Gamma \Vdash \Delta \quad \langle E \rangle}{\mathsf{close}\ \alpha\ .s :: \Phi \mid \Gamma, \alpha : T \Vdash \Delta \quad \left\langle T = \perp, E \right\rangle} \text{close}$$

$$\frac{s :: \Phi \mid \Gamma \Vdash \Delta \quad \langle E \rangle}{\mathsf{close}\ \alpha\ .s :: \Phi \mid \Gamma \Vdash \alpha : T, \Delta \quad \left\langle T = \perp, E \right\rangle} \text{close}$$

$$\frac{\phi :: \Phi \mid \phi \Vdash \phi}{\mathsf{halt}\ \alpha\ :: \Phi \mid \alpha : T \Vdash \phi \quad \left\langle T = \perp \right\rangle} \text{halt}$$

$$\frac{\phi :: \Phi \mid \phi \Vdash \phi}{\mathsf{halt}\ \alpha\ :: \Phi \mid \phi \Vdash \alpha : T \quad \left\langle T = \perp \right\rangle} \text{halt}$$

1.3 Split and Fork Command

split and fork commands come in a pair. $\mathsf{split}\ \ \alpha\ \ (\alpha_1, \alpha_2)$, splits a channel α into two channels α_1 and α_2. Depending on the polarity of α, the channels α_1 and α_2 are added to the respective channel contexts and are type inferred from the remainder of the process code. The protocol of the channel α is $\otimes$ or $\oplus$ of α_1 and α_2, depending on the polarity of α.

fork command spawns two processes from a single process. It assumes that one of the connected channels α splits into α_1 and α_2. Depending on the polarity of the channel α, the channels α_1 and α_2 are added to the output/input contexts of the two forked processes.The two processes are type inferred with their respective process code. This yields the protocol for the channels α_1 and α_2. The protocol of the channel α is obtained by doing $\otimes$ or $\oplus$ with the types of α_1 and α_2 depending on the polarity of α

Typing Rules for split-fork

$$\frac{s :: \Phi \mid \Gamma, \alpha_1 : T_1, \alpha_2 : T_2 \Vdash \Delta \quad \langle E \rangle}{\mathsf{split}\ \ \alpha\ \ (\alpha_1, \alpha_2).s :: \Phi \mid \Gamma, \alpha : T \Vdash \Delta \quad \left\langle \exists\, T_1, T_2 . T = T_1 \otimes T2, E \right\rangle}\ \text{split}$$

$$\frac{s :: \Phi \mid \Gamma \Vdash \alpha_1 : T_1, \alpha_2 : T_2, \Delta \quad \langle E \rangle}{\mathsf{split}\ \ \alpha\ \ (\alpha_1, \alpha_2).s :: \Phi \mid \Gamma \Vdash \alpha : T, \Delta \quad \left\langle \exists\, T_1, T_2 . T = T_1 \oplus T_2, E \right\rangle}\ \text{split}$$

$$\frac{s_1 :: \Phi \mid \Gamma_1 \Vdash \Delta_1, \alpha_1 : T_1 \quad \langle E_1 \rangle \quad s_2 :: \Phi \mid \Gamma_2 \Vdash \alpha_2 : T_2, \Delta_2 \quad \langle E_2 \rangle}{\begin{array}{l}\mathsf{fork}\ \alpha \\ \quad \mathsf{as} \left| \begin{array}{lll} \alpha_1 & \to & s_1 \\ \alpha_2 & \to & s_2 \end{array} \right. \end{array} :: \Phi \mid \Gamma_1, \Gamma_2 \Vdash \Delta_1, \alpha : T, \Delta_2 \quad \left\langle \exists\, T_1, T_2 . T = T_1 \otimes T_2, E_1, E_2 \right\rangle}\ \text{fork}$$

$$\frac{s_1 :: \Phi \mid \Gamma_1, \alpha_1 : T_1 \Vdash \Delta_1 \quad \langle E_1 \rangle \quad s_2 :: \Phi \mid \alpha_2 : T_2, \Gamma_2 \Vdash \Delta_2 \quad \langle E_2 \rangle}{\begin{array}{l}\mathsf{fork}\ \alpha \\ \quad \mathsf{as} \left| \begin{array}{lll} \alpha_1 & \to & s_1 \\ \alpha_2 & \to & s_2 \end{array} \right. \end{array} :: \Phi \mid \Gamma_1, \alpha : T, \Gamma_2 \Vdash \Delta_1, \Delta_2 \quad \left\langle \exists\, T_1, T_2 . T = T_1 \oplus T_2, E_1, E_2 \right\rangle}\ \text{fork}$$

1.4 plug command

plug command connects two processes with a certain of channels. To one process these channels act as the input channels and to the other they act as the output channels. In order to type infer these channels, they should be added to the output channel context of the first process and the input channel context of the second process. Now the two processes are type inferred in the enhanced channel context and channels' protocols obtained from the two processes are equated.

Typing Rule for plug
$$\frac{s_1 :: \Phi \mid \Gamma \Vdash \alpha_1 : T_1, \ldots, \alpha_n : T_n, \Delta \quad \langle E_1 \rangle \qquad s_2 :: \Phi \mid \Gamma, \alpha_1 : S_1, \ldots, \alpha_n : S_n \Vdash \Delta_2 \quad \langle E_2 \rangle}{\mathsf{plug}\ (\alpha_1, \ldots, \alpha_\mathsf{n})\ (\mathsf{s}_1, \mathsf{s}_2) :: \Phi \mid \Gamma \Vdash \Delta \quad \left\langle \exists \begin{array}{l} T_1, \ldots, T_n, \\ S_1, \ldots, S_n \end{array} . \begin{array}{l} T_1 = S_1, \ldots, T_n = S_n, \\ E_1, E_2 \end{array} \right\rangle}\ \text{plug}$$

1.5 pcase command

pcase or process case command is the concurrent world counter part of the case command from the sequential world. It gives us the ability to branch on the different constructors of a data type. Corresponding to each branch some process commands are run.

Let us defined a data type D, that has types variables A_1, …, A_k in the different constructors. $C_1 \ldots C_m$ are the different constructors of the data D. For constructor C_1 ,F_{11}, …, F_{1a} repesent the input type of the constructor.

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

Lets α rename the type variables A_1, …, A_k with fresh type variables, A'_1, …, A'_k in the different constructs of the definition above. 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.

$$\begin{array}{l} D_N = (\Lambda A_1, \ldots, A_k.\ D)\ A'_1, \ldots, A'_k \\ I_{11,N} = (\Lambda A_1, \ldots, A_k.\ I_{11})\ A'_1, \ldots, A'_k \\ \ \vdots \qquad\qquad\qquad \vdots \\ I_{mn,N} = (\Lambda A_1, \ldots, A_k.\ I_{mn})\ A'_1, \ldots, A'_k \end{array}$$

Typing Rules for process case

$$\frac{t : T_0\ \ \langle E_0\rangle,\ \ t_1 :: X_{11} : T_{11}, \ldots, X_{1a} : T_{1a}, \Phi \mid \Gamma \Vdash \Delta\ \ \langle E_1\rangle,\ \ \ldots,\ \ t_m :: X_{m1} : T_{m1}, \ldots, X_{mn} : T_{mn}, \Phi \mid \Gamma \Vdash \Delta\ \ \langle E_m\rangle}{\Gamma \vdash \begin{array}{l} \mathsf{pcase}\ t \\ \mathsf{of} \left| \begin{array}{lll} C_1 : X_{11}, \ldots, X_{1a} & \rightarrow & t_1 \\ \vdots & \vdots & \vdots \\ C_m : X_{m1}, \ldots, X_{mn} & \rightarrow & t_m \end{array} \right. \end{array} :: \Phi \mid \Gamma \Vdash \Delta \quad \left\langle \exists \begin{array}{l} T_0, \\ T_{11}, \ldots, T_{1a}, \\ \quad \vdots \\ T_{m1}, \ldots, T_{mn}, \\ A'_1, \ldots, A'_k \end{array} . \begin{array}{l} T_0 = D_{new} \\ T_{11} = F_{11,N}, \ldots, T_{11} = F_{1a,N} \\ \quad \vdots \\ T_{m1} = F_{m1,N}, \ldots, T_{mn} = F_{mn,N} \\ E_1, E_2, \ldots, E_m \end{array} \right\rangle}$$

1.6 run command

run command is used to call a process.The type of the process to be called is looked up from the symbol table. In the below representation, process p's type is comprised of s sequential types T_1, ... T_s, m input channels of type I_1, ... I_m and n output channels of type O_1, ... O_n. Like in the previous command case, the different constructs of the process type are renamed.

$$p : \forall A_1,\ \ldots,\ A_k.\ T_1, \ldots, T_s |\ I_1, \ldots, I_m \rightarrow\ O_1, \ldots, O_n$$

$$\begin{aligned}
T_{1,N} &= (\Lambda A_1, \ldots, A_k.\ T_1)\ A'_1, \ldots, A'_k \\
\vdots & \qquad\qquad \vdots \\
I_{1,N} &= (\Lambda A_1, \ldots, A_k.\ I_1)\ A'_1, \ldots, A'_k \\
\vdots & \qquad\qquad \vdots \\
O_{n,N} &= (\Lambda A_1, \ldots, A_k.\ O_n)\ A'_1, \ldots, A'_k
\end{aligned}$$

To type infer the run command, the sequential terms are type inferred in the sequential context Φ and the types obtained for these terms are equated with the types of the terms obtained from the process type looked up from the symbol table. The protocol of the input and the output channels are directly obtained from the process type.

Typing Rules for run

$$\frac{\Phi \vdash x_1 : X_1\langle E_1\rangle, \ldots, \Phi \vdash x_s : X_s\langle E_s\rangle}{\mathsf{p_1}\ (\mathsf{x_1}, \ldots \mathsf{x_s} | \alpha_1, \ldots, \alpha_\mathsf{m} \rightarrow \beta_1 \ldots \beta_\mathsf{m}) :: \Phi \mid \Gamma, \alpha_1 : P_1 \ldots \alpha_m : P_m \Vdash \beta_1 : Q_1, \ldots, \beta_n : Q_n, \Delta}\ \text{run}$$

$$\left\langle \exists \begin{array}{l} X_1, \ldots, X_s, \\ P_1, \ldots, P_m \\ Q_1, \ldots, Q_n, \\ A'_1, \ldots, A'_k \end{array} . \begin{array}{l} X_1 = S_{1,N}, \ldots, X_s = T_{s,N}, \\ P_1 = I_{1,N}, \ldots, P_m = I_{m,N}, \\ Q_1 = O_{1,N}, \ldots, Q_n = O_{n,N}, \\ E_1, \ldots, E_s \end{array} \right\rangle$$

1.7 id-neg Command

id command is used to equate two channels. When the the channels have different polarities then their channel protocols are equated other wise negation of one channel's protocol is equated with the protcol of the other channel.

Typing Rules for id-neg

$$\dfrac{}{\alpha \mid = \mid \beta :: \phi \mid \alpha : T_1 \Vdash \beta : T_2 \quad \langle T_1 = T_2 \rangle}\ \text{id}$$

$$\dfrac{}{\alpha \mid = \mid \beta :: \phi \mid \beta : T_2 \Vdash \alpha : T_1 \quad \langle T_1 = T_2 \rangle}\ \text{id}$$

$$\dfrac{}{\alpha \mid = \mid \textsf{neg}\ \beta :: \phi \mid \alpha : T_1, \beta : T_2 \Vdash \phi \quad \langle T_1 = Neg\ T_2 \rangle}\ \text{id}$$

$$\dfrac{}{\alpha \mid = \mid \textsf{neg}\ \beta :: \phi \mid \phi \Vdash \alpha : T_1, \beta : T_2 \quad \langle T_1 = Neg\ T_2 \rangle}\ \text{id}$$

1.8 hcase-hput Command

hput H on α command puts a handle/cohandle H on a channel α. The protocol of the handle H becomes the protocol of the channel α. The protocol of the channel α inferred from the remaining process commands should be equated to the input type of the handle H obtained from the symbol table.

hcase command branches on the handles of a protocol that a channel α has received. Corresponding to every handle certain process commands are run. The protocol of channel α is the protocol of which the cased handles are a part of. The process commands corresponding to every handle will produce a protocol for channel α. This protocol should be equated with the input type of that handle. The input type of a handle can be obtained from the symbol table.

Let P be a protocol definition with handles $H_1 \ldots H_n$. $T_1 \ldots T_n$ are the input types of the handles respectively. $A_1 \ldots A_k$ are the union of the type variables used in this protocol defintion.

$$\begin{array}{lllll} \forall A_1,\ \ldots,\ A_k.\ protocol\ P \implies C = & H_1 : T_1 & \implies & C \\ & \vdots & \vdots & \vdots \\ & H_n : T_n & \implies & C \end{array}$$

We α rename the input type of the handles, T_1, …, T_n and protocoll P with fresh type variables.

$$P_N = (\Lambda A_1, \ldots, A_k.\ P)\ A'_1, \ldots, A'_k$$
$$T_{i,N} = (\Lambda A_1, \ldots, A_k.\ T_i)\ A'_1, \ldots, A'_k$$

Typing Rules for hput-hcase

$$\frac{s :: \Phi \mid \Gamma, \alpha : S_1 \Vdash \Delta \ \ \langle E \rangle}{\text{hput H on } \alpha\ .s :: \Phi \mid \Gamma, \alpha : T \Vdash \Delta \ \ \left\langle \exists\, S_1, A'_1, \ldots, A'_k\ \ .T = P_N,\ S_1 = T_{1,N}\ ,\ \ E \right\rangle}\ \text{hput}$$

$$\frac{c_1 :: \Phi \mid \Gamma, \alpha : S_1 \Vdash \Delta \ \ \langle E_1 \rangle,\ \ \ldots,\ \ c_n :: \Phi \mid \Gamma, \alpha : S_n \Vdash \Delta \ \ \langle E_n \rangle}{\begin{array}{l}\text{hcase } \alpha \\ \text{of} \left| \begin{array}{lll} H_1 & \to & c_1 \\ \vdots & \vdots & \vdots \\ H_n & \to & c_n \end{array} \right. :: \Phi \mid \Gamma, \alpha : T \Vdash \Delta \end{array} \ \ \left\langle \exists \begin{array}{l} A'_1, \ldots, A'_k, \\ S_1, \ldots, S_n \end{array} .\ \begin{array}{l} T_0 = P_N \\ S_1 = T_{1,N}, \ldots, S_n = T_{n,N} \\ E_1, \ldots, E_n \end{array} \right\rangle}\ \text{hcase}$$

1.9 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 variables or constructor patterns. For the ease of describing the rule, the patterns have been divided into two cases.

- **Function with variables as input argument** (fdefn_1) - To type infer a 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 of the function defintion is type inferred.The input type of the function is the type of the input arguments (variables in this case) and the output type is the type of the term.
- **Function with constructor patterns as input arguments** (fdefn_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 case when the function take n constructor patterns as input. The input type of the function is the data type of the constructors of constructor pattern and the output type of the function is the type of the terms on the right hand side of the constructor. A well typed function defintion will have the terms corresponding to all the branches of the same type.

Typing rule for function defns with variable patterns
$$\frac{\Gamma, x_1 : I_1,\ \ldots,\ x_m : I_m \vdash t : O \ \ \langle E \rangle}{x_1, \ldots, x_m \ \ \rightarrow \ \ t \ \ : T \ \ \left\langle \exists\, I_1, \ldots, I_m \,.\ \begin{array}{l} T = (I_1, \ldots, I_m) \ \rightarrow \ O, \\ E \end{array} \right\rangle} \ \text{fdefn}_1$$