

Typing Rules for Concurrent MPL

March 10, 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 listed 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** $x \alpha$ 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** $t \alpha$ puts a sequential 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

$$\begin{array}{c}
\frac{s :: x : X, \Phi \mid \Gamma, \alpha : S \Vdash \Delta \quad \langle E \rangle}{\text{get } x \ \alpha . s :: \Phi \mid \Gamma, \alpha : T \Vdash \Delta \quad \left\langle \exists X, S. T = \text{Put}(X, S), E \right\rangle} \text{get} \\
\\
\frac{s :: x : X, \Phi \mid \Gamma \Vdash \alpha : S, \Delta \quad \langle E \rangle}{\text{get } x \ \alpha . s :: \Phi \mid \Gamma \Vdash \alpha : T, \Delta \quad \left\langle \exists X, S. T = \text{Get}(X, S), E \right\rangle} \text{get} \\
\\
\frac{\Phi \vdash t : X \quad \langle E_1 \rangle \quad s :: \Phi \mid \Gamma, \alpha : S \Vdash \Delta \quad \langle E_2 \rangle}{\text{put } t \ \alpha . s :: \Phi \mid \Gamma, \alpha : T \Vdash \Delta \quad \left\langle \exists X, S. T = \text{Get}(X, S), E_1, E_2 \right\rangle} \text{put} \\
\\
\frac{\Phi \vdash t : X \quad \langle E_1 \rangle \quad s :: \Phi \mid \Gamma \Vdash \alpha : S, \Delta \quad \langle E_2 \rangle}{\text{put } t \ \alpha . s :: \Phi \mid \Gamma \Vdash \alpha : T, \Delta \quad \left\langle \exists X, S. T = \text{Put}(X, S), E_1, E_2 \right\rangle} \text{put}
\end{array}$$

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

1.3 Split and Fork Command

split and **fork** commands come in a pair. **split** α (α_1, α_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	
$\frac{s :: \Phi \mid \Gamma, \alpha_1 : T_1, \alpha_2 : T_2 \Vdash \Delta \quad \langle E \rangle}{\text{split } \alpha \quad (\alpha_1, \alpha_2). s :: \Phi \mid \Gamma, \alpha : T \Vdash \Delta \quad \left\langle \exists T_1, T_2. T = T_1 \otimes T_2, E \right\rangle} \text{ split}$	
$\frac{s :: \Phi \mid \Gamma \Vdash \alpha_1 : T_1, \alpha_2 : T_2, \Delta \quad \langle E \rangle}{\text{split } \alpha \quad (\alpha_1, \alpha_2). s :: \Phi \mid \Gamma \Vdash \alpha : T, \Delta \quad \left\langle \exists T_1, T_2. T = T_1 \oplus T_2, E \right\rangle} \text{ split}$	
$\frac{s_1 :: \Phi \mid \Gamma_1 \Vdash \Delta_1, \alpha_1 : T_1 \quad \langle E_1 \rangle \quad s_2 :: \Phi \mid \Gamma_2 \Vdash \alpha_2 : T_2, \Delta_2 \quad \langle E_2 \rangle}{\text{fork } \alpha \quad \text{as } \left \begin{array}{l} \alpha_1 \rightarrow s_1 \\ \alpha_2 \rightarrow s_2 \end{array} \right. :: \Phi \mid \Gamma_1, \Gamma_2 \Vdash \Delta_1, \alpha : T, \Delta_2 \quad \left\langle \exists T_1, T_2. T = T_1 \otimes T_2, E_1, E_2 \right\rangle} \text{ fork}$	
$\frac{s_1 :: \Phi \mid \Gamma_1, \alpha_1 : T_1 \Vdash \Delta_1 \quad \langle E_1 \rangle \quad s_2 :: \Phi \mid \alpha_2 : T_2, \Gamma_2 \Vdash \Delta_2 \quad \langle E_2 \rangle}{\text{fork } \alpha \quad \text{as } \left \begin{array}{l} \alpha_1 \rightarrow s_1 \\ \alpha_2 \rightarrow s_2 \end{array} \right. :: \Phi \mid \Gamma_1, \alpha : T, \Gamma_2 \Vdash \Delta_1, \Delta_2 \quad \left\langle \exists T_1, T_2. T = T_1 \oplus T_2, E_1, E_2 \right\rangle} \text{ fork}$	

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	
$\frac{s_1 :: \Phi \mid \Gamma \Vdash \alpha_1 : T_1, \dots, \alpha_n : T_n, \Delta \quad \langle E_1 \rangle \quad s_2 :: \Phi \mid \Gamma, \alpha_1 : S_1, \dots, \alpha_n : S_n \Vdash \Delta_2 \quad \langle E_2 \rangle}{\text{plug } (\alpha_1, \dots, \alpha_n) (s_1, s_2) :: \Phi \mid \Gamma \Vdash \Delta \quad \left\langle \begin{array}{l} \exists \quad T_1, \dots, T_n, \quad T_1 = S_1, \dots, T_n = S_n, \\ S_1, \dots, S_n \quad . \quad E_1, E_2 \end{array} \right\rangle} \text{plug}$	

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.

Lets defined a data type D, that has types variables A_1, \dots, A_k in the different constructors. $C_1 \dots C_m$ are the different constructors of the data D. For constructor $C_1, F_{11}, \dots, F_{1a}$ represent the input type of the constructor.

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

Lets α rename the type variables A_1, \dots, A_k with fresh type variables, A'_1, \dots, 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.

$$\begin{aligned} D_N &= (\lambda A_1, \dots, A_k. D) A'_1, \dots, A'_k \\ I_{11,N} &= (\lambda A_1, \dots, A_k. I_{11}) A'_1, \dots, A'_k \\ &\vdots \\ I_{mn,N} &= (\lambda A_1, \dots, A_k. I_{mn}) A'_1, \dots, A'_k \end{aligned}$$

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

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, \dots, T_s , m input channels of type I_1, \dots, I_m and n output channels of type O_1, \dots, O_n . Like in the previous command **case**, the different constructs of the process type are renamed.

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

$$T_{1,N} = (\Lambda A_1, \dots, A_k. T_1) A'_1, \dots, A'_k$$

$$\vdots \quad \quad \quad \vdots$$

$$I_{1,N} = (\Lambda A_1, \dots, A_k. I_1) A'_1, \dots, A'_k$$

$$\vdots \quad \quad \quad \vdots$$

$$O_{n,N} = (\Lambda A_1, \dots, A_k. O_n) A'_1, \dots, A'_k$$

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	
$\frac{\Phi \vdash x_1 : X_1 \langle E_1 \rangle, \dots, \Phi \vdash x_s : X_s \langle E_s \rangle}{\mathbf{p_1} (x_1, \dots, x_s \mid \alpha_1, \dots, \alpha_m \rightarrow \beta_1 \dots \beta_n) :: \Phi \mid \Gamma, \alpha_1 : P_1 \dots \alpha_m : P_m \Vdash \beta_1 : Q_1, \dots, \beta_n : Q_n, \Delta} \text{ run}$	
$\left\langle \exists \begin{array}{ll} X_1, \dots, X_s, & X_1 = S_{1,N}, \dots, X_s = T_{s,N}, \\ P_1, \dots, P_m & P_1 = I_{1,N}, \dots, P_m = I_{m,N}, \\ Q_1, \dots, Q_n, & Q_1 = O_{1,N}, \dots, Q_n = O_{n,N}, \\ A'_1, \dots, A'_k & E_1, \dots, E_s \end{array} \right\rangle$	

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 protocol of the other channel.

Typing Rules for id-neg	
$\frac{\alpha \mid = \mid \beta :: \phi \mid \alpha : T_1 \Vdash \beta : T_2 \quad \langle T_1 = T_2 \rangle}{\text{id}}$	
$\frac{\alpha \mid = \mid \beta :: \phi \mid \beta : T_2 \Vdash \alpha : T_1 \quad \langle T_1 = T_2 \rangle}{\text{id}}$	
$\frac{\alpha \mid = \mid \text{neg } \beta :: \phi \mid \alpha : T_1, \beta : T_2 \Vdash \phi \quad \langle T_1 = \text{Neg } T_2 \rangle}{\text{id}}$	
$\frac{\alpha \mid = \mid \text{neg } \beta :: \phi \mid \phi \Vdash \alpha : T_1, \beta : T_2 \quad \langle T_1 = \text{Neg } T_2 \rangle}{\text{id}}$	

1.8 hcase-hput Command

hput H on α command puts a handle 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 $H_1 \dots H_n$. $T_1 \dots T_n$ are the input types of the handles respectively. $A_1 \dots A_k$ are the union of the type variables used in this protocol definition.

$$\begin{aligned} \forall A_1, \dots, A_k. \text{protocol } P &\implies C = H_1 : T_1 \implies C \\ &\vdots \quad \vdots \quad \vdots \\ &H_n : T_n \implies C \end{aligned}$$

We α rename the input type of the handles, T_1, \dots, T_n and protocol P with fresh type variables.

$$\begin{aligned} P_N &= (\lambda A_1, \dots, A_k. P) A'_1, \dots, A'_k \\ T_{i,N} &= (\lambda A_1, \dots, A_k. T_i) A'_1, \dots, A'_k \end{aligned}$$

Typing Rules for hput-hcase	
$\frac{s :: \Phi \mid \Gamma, \alpha : S_1 \Vdash \Delta \quad \langle E \rangle}{\text{hput H on } \alpha . s :: \Phi \mid \Gamma, \alpha : T \Vdash \Delta \quad \left\langle \exists S_1, A'_1, \dots, A'_k . T = P_N, S_1 = T_{1,N}, E \right\rangle} \text{hput}$	
$\frac{c_1 :: \Phi \mid \Gamma, \alpha : S_1 \Vdash \Delta \quad \langle E_1 \rangle, \dots, c_n :: \Phi \mid \Gamma, \alpha : S_n \Vdash \Delta \quad \langle E_n \rangle}{\text{hcase } \alpha} \text{hcase}$	
of	$\left \begin{array}{l} H_1 \rightarrow c_1 \\ \vdots \\ H_n \rightarrow c_n \end{array} \right. :: \Phi \mid \Gamma, \alpha : T \Vdash \Delta \quad \left\langle \exists \begin{array}{l} A'_1, \dots, A'_k, \\ S_1, \dots, S_n . \end{array} \begin{array}{l} T_0 = P_N \\ S_1 = T_{1,N}, \dots, S_n = T_{n,N} \\ E_1, \dots, E_n \end{array} \right\rangle$