Res. Rep. Fac. Eng. Mie Univ., Vol. 16, pp. 79-88 (1991)

Original Paper

Denotational Semantics of a Language for Asynchronously Communicating Processes

Takashi NASU *, Michio OYAMAGUCHI
and Yoshikatsu OHTA
(Department of Information Processing)

(Received September 16,1991)

Abstract

We have designed a programming language for parallel processing which have the following fundamental facilities: parallel execution, asynchronously communication with channels, nondeterministic selection, dynamic creation of processes, etc. And we have presented a simple and highly abstract denotational semantics for this language.

1 Introduction

Recently, for the need of higher computational power, parallel processing has been paid much attention to. So far, many programming languages suitable for parallel processing has been proposed and the semantics of parallel processing language has been studied actively. As the method of the semantic description, there are operational methods, axiomatic methods, denotational methods, etc. But, parallel processing languages have not only interaction of processes but also nondeterminism. So, the semantic description is difficult in comparison with those for usual programming languages. In particular, as Brock, Ackerman and et al. pointed out, in the semantic description of the parallel processing languages which has the function of the nondeterminism, one need to represent the temporal relation of the communication any way. But, describing the temporal relation too explicitly means refering to the details of the implementation. So, that is not desirable to give the highly abstractive semantics.

We have designed a programming language for parallel processing for the purpose of giving a simple and highly abstract denotational semantics for a practical parallel processing language. This language has the following fundamental facilities: parallel execution, asynchronously communication with channels, nondeterministic selection, dynamic creation of processes, etc.

In this report, we firstly describe the parallel processing language we have proposed. And then we present the semantics for this language giving the semantic functions.

2 The Language Specification

The purpose for our research is to describe the semantics for a practical parallel processing language simply and abstractly. But, practical parallel processing languages already exisited have too many functions, so it is difficult to give them their semantics simply. Therefore, we have studied the subset of the parallel processing language whose functions are restricted to the essential ones for the semantic description.

This language have the facilities of asynchronously communication with channels between processes, process calls in parallel and nondeterministic selection. This language have been extended adding the facilities of dynamic process creation, declaration of local variables, etc to the language which was given the semantics by Broy[3]. In figure 1, we show the syntax of this language in BNF notation.

Where, $<$ var $-$ id $>$ is a variable name declared in the block except a channel type. $<$ inpch $-$ id $>$ and $<$ outch $-$ id $>$ is a channel variable name for input and output, respectively. And $<$ process $-$ id $>$ is a process variable name and $<$ id $>$ is a name. ϵ is an empty statement.

"Program" consists of a sequence of process declarations and a block representing the initial process. Where, $<$ process declaration $>;\cdots$ is a sequence of process declarations. "Process declaration" consists of a formal

* Mitsubishi Electric Corporation

< program > ::= **program** < id > < parm > ;
< process declaration >;$\cdots$;< block >

< process declaration > ::= **process** < id > < parm > ; < block > **endprocess**

< parm > ::= ( ) | (< parmlist >)

< parmlist > ::= < id >,$\cdots$: < type >
| < id >,$\cdots$: < type > ; < parmlist >

< statement > ::= ϵ | **begin** < statement >;$\cdots$ **end**
| < var-id > := < expression >
| **if** < expression > **then** < statement >
| **if** < expression > **then** < statement > **else** < statement >
| **while** < expression > **do** < statement >
| **repeat** < statement > **until** < expression >
| < inpch-id >? < var-id >
| < outch-id >! < expression >
| **chan** < inpch-id >$\leftarrow$< outch-id >:< statement >
| < process-id > < parm >
| < statement > // < statement > // $\cdots$
| < statement > [] < statement > [] $\cdots$

Figure 1: Overview of the syntax

parameter declaration and a block and "block" consists of a local variable declaration and a sequence of statements enclosing with **begin** and **end**.

This language involves ordinary compound statements, assignment statements, conditional statements and repetitive statements. Besides, it involves channel connect statements, process call statements, parallel execute statements and channel input/output statements for the construction of process network and the communication between processes. Moreover, there is a nondeterministic select statement; e.g. the execution of the statement S_1 [] S_2 is done as follows: first S_1 or S_2 is selected nondeterministicly and then the selected statement is executed. Which statement is selected depend on the scheduling policy of an implementation.

Process call statement creates process instances dynamically and those created processes are executed. To allow to create multiple instances for same process, we introduce process variables. Therefore, it is necessary to declare the association a process variable with a process name in the declaration part. For example, we declare *merge* as a process variable of MERGE2, *out1* as OUTLIST and *plus* as PLUS1 in the initial process of the sample program(Fig. 2).

When a process creates other processes, the execution of the process is waited until all created process are terminated. In this point, a process call statement likes a procedure call of a usual programming languages. But, the former can create multiple process simultaneously and execute them concurrently combining with a parallel execute statement. In the sample program, using process variables declared, a process call is done concurrently. Then, three processes are created and executed concurrently.

In this language, arguments are passed by addresses to a process. Therefore, expressions cannot be specified as actual parameters.

We have following restriction with variables: there is no shared variable used in each statements of a parallel execute statement, channel variables used at a channel connect statement must be declared locally in that process, and so on.

2.1 Processes

A process is a processing unit which can execute computation independently. It is defined by a process declaration and a new instance is created by executing a process call statement. The created process executes the statement

```
program FeedBack();

process MERGE2(c,ei:inpch; zo:outch);
var x:integer;
beginc?x [] ei?x; zo!x;
    c?x [] ei?x; zo!x end
endprocess;

process OUTLIST( zi:inpch; d,wo:outch);
var x,y:integer; bf begin zi?x; zi?y; d!x; wo!x end
endprocess;

process PLUS1(wi:inpch; eo:outch);
var x:integer; begin wi?x; eo!(x+1) end
endprocess;

(* main = initial process *)
var c,ei,wi,zi:inpch; d,eo,wo,zo:outch;
   merge:MERGE2; out1:OUTLIST; plus:PLUS1;
begin chan ei ← eo: chan wi ← wo: chan zi ← zo:
    merge(c,ei,zo) // out1(zi,d,wo) // plus(wi,e0)
end.
```

Figure 2: A Sample Program

in the process body and disappeared on the termination of the execution. A process can have local variables but no shared memories. The communications with other processes are performed by message passing through channels. A process have execute or wait status. When a process request a channel but a message has not yet been arrived, the process become wait status. Lately, when the message will be arrived, the process status will be changed from wait status to execute status, then the process will be resumed. The process network can be created dynamically, and the created process can create the network for child processes, too. When the network for child processes is created, the parent process becomes wait status and when all child processes will terminate, the parent process will return execute status.

A parent process can pass a channel connected with other process as a parameter to a child process. A child process can communicate to the external world using this channel.

2.2 Channels

The communication between processes is the one-way asynchronous communication using channels(communication lines). We suppose that a channel has an infinite buffer. The communication line between processes can establish when one specifies the connection between an input channel variable of one process(P_1) and an output channel variable of another process(P_2). Thereafter through this communication line a message sending from P_2 to P_1 becomes possible. Besides, since the number of input or output channels which a process can use is not limited, one can construct a complex network freely. But, one cannot connect two or more input(or output) channel variables to an output (or input) channel variables.

The message sending through a communication line arrives at a receiver within a finite time. That is, the time when a receiver gets a message depend on an implementation. The message sequence sent through a communication line is received in the order that the message has been sent.

For example, let P and Q be processes, c and d be an input and output variables, respectively. When the following statement is executed,

chan $c \leftarrow d : P(c) // Q(d)$

processes P and Q are created simultaneously and then executed concurrently. Besides, with the channel connection, an output to the channel d is passed to the channel c and then a communication between processes is done.

$\rho \in \mathtt{Lenv} = [\mathtt{id} \to \mathtt{Loc}]$

$\sigma \in \mathtt{State} = [\mathtt{Location} \to \mathtt{Value}]$

$\mathrm{g} \in \mathtt{Genv} = [\mathtt{id} \to \mathtt{Proc\_func}]$

$\mathtt{Proc\_func} = \cup_{m \geq 0} \mathtt{Location}^m \to (\mathtt{Lenv} \times \mathtt{State}) \to \mathcal{P}(\mathtt{Lenv} \times \mathtt{State})]$

$\mathcal{P}\mathrm{rg} \in [\mathtt{Program} \to \mathtt{Genv} \to (\mathtt{Lenv} \times \mathtt{State}) \to \mathcal{P}(\mathtt{Lenv} \times \mathtt{State})]$

$\mathcal{D}\mathrm{p} \in [\mathtt{Process\_decl} \to \mathtt{Genv} \to \mathtt{Genv}]$

$\mathcal{B}\mathrm{l} \in [\mathtt{Block} \to \mathtt{Genv} \to (\mathtt{Lenv} \times \mathtt{State}) \to \mathcal{P}(\mathtt{Lenv} \times \mathtt{State})]$

$\mathcal{B} \in [\mathtt{Var\_decl} \to (\mathtt{Lenv} \times \mathtt{State}) \to (\mathtt{Lenv} \times \mathtt{State})]$

$\mathcal{C}, \mathcal{C}\mathrm{c} \in [\mathtt{Statement} \to \mathtt{Genv} \to (\mathtt{Lenv} \times \mathtt{State}) \to \mathcal{P}(\mathtt{Lenv} \times \mathtt{State})]$

$\mathcal{E} \in [\mathtt{Expression} \to \mathtt{Lenv} \to \mathtt{State} \to \mathtt{Value}]$

Figure 3: Semantic Regions

3 The Formal Semantics Description

In our semantic description, when a process becomes wait status in a channel communication, the place is memoried and then when a message is arrived at the channel, the operation after the place is executed. This looks like an operational method, but it is necessary to memory the minimal informations such as the place where the wait occurs. To describe these information in the method of a denotational semantics, we define the status whether an input wait is occured and the memory of the place where a wait occurs in the semantic regions. Beside, we introduce some auxiliary functions(predicates) to handle these informations. We show these informations and auxiliary functions in the appendix.

The semantic regions used in semantic description is shown at Fig. 3. Here, "Global environment(Genv)" maps a process name to its function. "Local environment(Lenv)" maps a variable name to its location and "state(State)" maps a location to its value. A semantic function Proc_func maps argument locations and a pair of local environment and status to a power set $\mathcal{P}(\mathtt{Lenv} \times \mathtt{State})$. We represent the multiple result of a nondeterministic selection as a power set. Here, the power set is similar to the "Power-domain"[4,5]. The set of states given by semantic function of a process is coresponding to all computational configurations collected in the parallel processing model.

The semantic function of "statement" is a function which takes a statement, a local environment and a pair of a local environment and a state as arguments and return its power set. When a process declaration is given, the semantic function of the process declaration embeds a pair of its process name and its semantic function into a global environment.

The semantics of non-terminating program is given a "bottom" function in the standpoint of the denotational semantics. In this report, we consider an another function as the bottom function, which outputs the values of the output channel as it is at least, like the way of Broy-Lengauer[3]. Because, for example, if we consider a program which outputs infinite sequence of "1" to a output channel, we want to give a semantics which the program continue outputing "1", not no semantics for the reason of non-termination.

In the following, we present the semantic functions by cases of program constructs.

3.1 Sematic function of a program $\mathcal{P}rg$

The sematic function of a program $\mathcal{P}rg$ is defined as follows: First, changing global environment by the sequence of process declarations, and then using this global environment the semantic function of the block of the initial process is applied to given local environment and status.

$\mathcal{P}rg[\![\mathtt{program}\ \Delta_1; \cdots; \Delta_m; \Theta\ .\ ]\!]_g(\rho, \sigma) = \mathcal{B}l[\![\Theta]\!](\mathcal{D}p[\![\Delta_1; \cdots; \Delta_m]\!]g)(\rho, \sigma)$
where, $\Delta_1; \cdots; \Delta_m$ are process declarations , Θ is a block, $g \in \mathtt{Genv}$, $\rho \in \mathtt{Lenv}$, $\sigma \in \mathtt{State}$

3.2 Semantic function of a process $\mathcal{D}p$

The semantic function of a process $\mathcal{D}p$ embeds the association of the functions which gives meaning to each processes and process names into global environment using the sequence of process declarations which defines processes in the program. The meaning of each processes is the meaning of the block of each process when a location given as an actual argument is associated to a formal parameter. Because it is allowed that each process calls each other, we give the meaning using the fix-point.

$$\mathcal{D}p[\![\Delta_1;\cdots;\Delta_m]\!]_g = g[\gamma_1/\mathrm{P}_1,\ldots,\gamma_m/\mathrm{P}_m]$$
where Δ_i = **process** $\mathrm{P}_i(a_i);\Theta_i$ **endprocess** $\quad 1 \le i \le m$

$$(\gamma_1,\ldots,\gamma_m) = fix(\lambda(\gamma_1,\ldots,\gamma_m) : (\lambda(\rho_1,\sigma_1) : \lambda\mu_1 : \mathcal{B}l[\![\Theta_1]\!]_{g'}(\rho_1[\mu_1/a_1],\sigma_1)\ ),$$
$$\cdots\ ,$$
$$(\lambda(\rho_m,\sigma_m) : \lambda\mu_m : \mathcal{B}l[\![\Theta_m]\!]_{g'}(\rho_m[\mu_m/a_m],\sigma_m)\ )\ )$$
$$g' = g[\gamma_1/\mathrm{P}_1,\ldots,\gamma_m/\mathrm{P}_m]$$

3.3 Semantic function of a block $\mathcal{B}l$

The semantic function of a block $\mathcal{B}l$ is the result of the application of the semantic function of the block after applying the semantics function of the variable declaration. But, if $\sigma.wait$ is *true*, it means that this block is executed once and an input wait is occured. Then, it is regarded that the processing of the variable declaration is done. Where, $\sigma.wait$(that is, $\sigma(wait)$) represents whether an input wait is occured until then(that is, $\sigma.wait \in \{\perp, true, false\}$).

In this reason, we don't apply the variable declaration and apply given local environment and status to the semantic function of the statement.

$$\mathcal{B}l[\![\textbf{var } \delta;\Gamma]\!]_g = (\lambda(\rho,\sigma) : \mathcal{C}c[\![\Gamma]\!]_g(\rho',\sigma'))$$
where (ρ',σ') = (**IF** $\sigma.wait = false$ **THEN** $\mathcal{B}$[**var** δ](ρ,σ) **ELSE** (ρ,σ))

δ is a < `parmlist` > , Γ is a statement sequence enclosing with **begin** and **end**

3.4 Semantic function of a variable declaration $\mathcal{B}$

The variable declaration allocates the memory location for local variables and updates the local environment. A location is extracted from the unused locations $\sigma.l_0$ managed by the special location l_0 and then it is assigned to a local variable of the created process. Where, $\sigma.l_0$ represents the set of the unsed locations on the state σ (that is, $\sigma.l_0 \in \mathcal{P}(\mathtt{Location})$). When a process is created and the allocation of memory locations for that local variables is needed, we use $\sigma.l_0$ for managing the locations.

$$\mathcal{B}[\![\textbf{var } \delta_1;\delta_2]\!] = (\lambda(\rho,\sigma) : \mathcal{B}[\![\textbf{var } \delta_2]\!](\mathcal{B}[\![\textbf{var } \delta_1]\!](\rho,\sigma))\ )$$

$$\mathcal{B}[\![\textbf{var } \mathrm{x}_1,\mathrm{x}_2,\ldots,\mathrm{x}_n{:}\tau]\!] = (\lambda(\rho,\sigma) : \mathcal{B}[\![\textbf{var } \mathrm{x}_2,\ldots,\mathrm{x}_n{:}\tau]\!](\mathcal{B}[\![\textbf{var } \mathrm{x}_1{:}\tau]\!](\rho,\sigma))\ )$$

where, δ_1, δ_2 are < `parmlist` > , τ is a type

$$\mathcal{B}[\![\textbf{var } \mathrm{x}{:}\tau]\!](\rho,\sigma) = (\lambda l_x : (\rho[l_x/\mathrm{x}],\ \sigma[removeloc(\sigma.l_0,\ l_x)/l_0, \perp/l_x])\ )(getloc(\sigma.l_0))$$

where, τ is a type except a process name , `inpch`, `outch`

$$\mathcal{B}[\![\textbf{var } \mathrm{c} : \mathtt{inpch}]\!](\rho,\sigma) = (\lambda(l_d,l_w,l_l,l_c) : (\rho\ [l_d/\mathrm{c},\ l_w/\mathrm{c.wait},\ l_l/\mathrm{c.waitloc},\ l_c/\mathrm{c.connect}],$$
$$\sigma[removeloc(\ \sigma.l_0,(l_d+l_w+l_l+l_c))/l_0,$$
$$\epsilon/l_d,\ false/l_w,\ \epsilon/l_l,\ false/l_c])\ )$$
$$(getloc^4(\sigma.l_0))$$

$$\mathcal{B}[\![\textbf{var } \mathrm{d} : \mathtt{outch}]\!](\rho,\sigma) = (\lambda l_d : (\rho[l_d/\mathrm{d}],\ \sigma[removeloc(\sigma.l_0,ld)/l_0,\ \epsilon/l_d])\ )(getloc(\sigma.l_0))$$

$$\mathcal{B}[\![\textbf{var } \mathrm{p}{:}\tau_p]\!](\rho,\sigma) = (\lambda(l_d,l_w,l_l,l_c) : (\rho\ [l_d/\mathrm{p},\ l_w/\mathrm{p.wait},\ l_l/\mathrm{p.waitloc},\ l_e/\mathrm{p.env}],$$
$$\sigma[removeloc(\sigma.l_0,(l_d+l_w+l_l+l_e))/l_0,\ \tau_p/l_d,\ false/l_w,\epsilon/l_l,\ \perp/l_e])\ )$$
$$(getloc^4(\sigma.l_0))$$

where, τ_p is a process name.

3.5 Semantic function of a statement Cc, C

Cc is a semantic function including the control of execution when a input wait is occured in the communication. When $\sigma.wait$ is false, that is, an input wait is not occured till then, or when a input wait is occured within the statement, Cc applies the semantic function of the statement C. Otherwise, Cc returns a set contained only the pair of original local environment and status. This means that the statement is not executed and skipped.

$$Cc[\![\mathrm{S}]\!]_g = (\lambda(\rho,\sigma) : \textbf{IF } \sigma.wait = false \vee \neg SKIP_\rho(\sigma, LABEL(\mathrm{S}))$$
$$\textbf{THEN } C[\![\mathrm{S}]\!]_g(\rho,\sigma)$$
$$\textbf{ELSE } \{\ (\rho,\sigma)\ \}\ )$$

$SKIP_\rho(\sigma, Lset) = \neg(\exists \mathrm{c} \in \mathrm{ICI} \cup \mathrm{PSI} : \sigma.\rho.\mathrm{c.wait} = true\ \wedge\ \sigma.\rho.\mathrm{c.waitloc} \in Lset\ )$
(ICI, PSI are an input channel variable a set of a process variable name, respectively
$SKIP_\rho(\sigma, LABEL(\mathrm{S}))$ is true = no wait status in statement S)

In the following, we explain the semantic function of each statement.

3.5.1 Empty statement

An empty statement don't cause any changes, then return a set contained only original local environment and state.

$$C[\![\ \texttt{begin end}\ ]\!]_g = (\lambda(\rho,\sigma) :\ \{\ (\rho,\sigma)\ \}\ )$$

3.5.2 Compound statement

The statement sequence enclosing with **begin** and **end** is executed sequentially. Then, we apply the result of the application of the semantic function of first statement to the semantic function of the following statements. Because the result of the semantic function of a statement is a set of a pair of local environment and state, we use functional composition $\circ$.

$$C[\![\ \texttt{begin}\ \mathrm{S}_1;\mathrm{S}_2;\cdots\ \mathrm{S}_n\ \texttt{end}\ ]\!]_g = Cc[\![\mathrm{S}_1]\!]_g \circ C[\![\ \texttt{begin}\ \mathrm{S}_2;\cdots\ \mathrm{S}_n\ \texttt{end}\ ]\!]_g$$
where, $p \circ q = (\lambda(\rho,\sigma) : \{\ (\rho'',\sigma'')\ \mid \exists(\rho',\sigma') \in p(\rho,\sigma)\ , (\rho'',\sigma'') \in q(\rho',\sigma')\ \})$
$p, q, p \circ q \in [(\mathrm{Lenv} \times \mathrm{State}) \rightarrow \mathcal{P}(\mathrm{Lenv} \times \mathrm{State})]$

3.5.3 If statement

If $\sigma.wait$ is false, that is, there is no wait on the execution of the program till then, we evaluate the conditinal expression as a usual language, then execute the statement following **then** or **else** at the truth of that expression. But, when an input wait is occured, we select the statement where an input wait is occured regardless of the truth. Because we define that the semantic function C is executed only when an input wait is occured in the **if** statement by the semantic function Cc, we can select the statement only from the truth of $\neg SKIP_\rho(\sigma, LABEL(\mathrm{S}_1))$.

$$C[\![\texttt{if}\ E\ \texttt{then}\ \mathrm{S}_1\ \texttt{else}\ \mathrm{S}_2]\!]_g = (\lambda(\rho,\sigma) : \textbf{IF}\ (\ \sigma.wait = false \wedge \mathcal{E}[\![E]\!]_\rho.\sigma = true)$$
$$\vee\neg SKIP_\rho(\sigma, LABEL(\mathrm{S}_1))$$
$$\textbf{THEN}\ C[\![\mathrm{S}_1]\!]_g(\rho,\sigma)$$
$$\textbf{ELSE}\ C[\![\mathrm{S}_2]\!]_g(\rho,\sigma)\ )$$

3.5.4 Assignment statement

The state σ is updated so that the value of the location $\rho.\mathrm{x}$ of the variable name (x) in the left hand side becomes the evaluated value of the expression in the right hand side $\mathcal{E}[\![E]\!]_\rho.\sigma$.

$$C[\![\mathrm{x} := E]\!]_g = (\lambda(\rho,\sigma) : \{\ (\rho, \sigma[\mathcal{E}[\![E]\!]_\rho.\sigma / \rho.\mathrm{x}])\ \})$$

3.5.5 Channel input statement

If the content of the input channel is empty, an input wait is occured. In this time, we make σ.wait true, and record the lable attached to this channel input statement into the field of the input channel variable $\sigma.\rho.$c.waitloc. Moreover, we update the state so that $\sigma.\rho.$c.wait becomes true which indicates that waitloc is valid. If the input channel has datum, in this case even if an input wait is occured, it is cancelled, we make σ.wait and $\sigma.\rho.$c.wait false. And then we store the first data of the content of the input channel to the variable x and make the content of the channel the remaining data sequence.

$$\begin{aligned}\mathcal{C}[\![L_i:\ \mathrm{c}\ ?\ \mathrm{x}]\!]_g = (\lambda(\rho,\sigma):\ &\textbf{IF}\ \ \sigma.\rho.\mathrm{c} = \epsilon\\ &\textbf{THEN}\ \ \{\ (\rho, \sigma[\mathit{true}/\mathit{wait},\ \mathit{true}/\rho.\mathrm{c.wait},\ L_i/\rho.\mathrm{c.waitloc}])\ \}\\ &\textbf{ELSE}\ \ \{\ (\rho,\ \sigma[\mathit{false}/\mathit{wait},\ \ \mathit{false}/\rho.\mathrm{c.wait},\\ &\qquad\qquad \mathit{first}(\sigma.\rho.\mathrm{c})/\rho.\mathrm{x},\ \mathit{rest}(\sigma.\rho.\mathrm{c})/\rho.\mathrm{c}])\ \}\ )\end{aligned}$$

3.5.6 Channel output statement

Same as the previous assigment statement, but the updated value is a sequence appended the expression value $\mathcal{E}[\![E]\!]_\rho.\sigma$ to the current content of the output channel $\sigma.\rho.$d.

$$\mathcal{C}[\![\mathrm{d}\ !\ E]\!]_g = (\lambda(\rho,\sigma):\ \{\ (\rho, \sigma[\mathit{append}(\sigma.\rho.\mathrm{d},\ \mathcal{E}[\![E]\!]_\rho.\sigma)/\rho.\mathrm{d}]\ \})$$

3.5.7 Channel connect statement

First, we apply the semantic function of the statement. And then we append the data obtained in the output channel to the input channel. If there is an input wait in the execution of the statement, we execute the statement again.

When an input wait is occured in the channel connection outside of this channel connect statement, we cannot cancel the input wait by the repetition which gives the semantics of the inside channel connect statement.

For this reason, we prepare the field connect which indicates the connection to the input channel, and we make connect true in the entrance of the repetitive loop and false in the exit of the loop. And even if there is an input wait, if there are a channel connection outside, that is, the channel whose connect field is true, we exit the loop. To check the existence of the channel connected outside, we use the auxiliary function *Isconnect*.

The repetition is define by a recursive function. The semantic function of the channel connect statement is a fix-point of this recursive function.

$$\begin{aligned}\mathcal{C}[\![\mathrm{chan\ c}\ \leftarrow\ \mathrm{d}:\ \mathrm{S}]\!]_g = \mathit{fix}(\lambda F:\ &(\lambda(\rho_1,\sigma_1):\mathcal{C}c[\![\mathrm{S}]\!]_g(\rho_1,\ \sigma_1[\mathit{true}/\rho_1.\mathrm{c.connect}])\ )\circ\\ &(\lambda(\rho_2,\sigma_2):\ \textbf{IF}\ \ (\sigma_2'.\mathit{wait} = \mathit{false}) \vee \mathit{Isconnect}(\rho_2,\sigma_2')\ )\\ &\qquad\qquad \textbf{THEN}\ \ \{\ (\rho_2,\sigma_2')\ \}\\ &\qquad\qquad \textbf{ELSE}\ \ F(\rho_2,\sigma_2')\ )\ )\end{aligned}$$

where $\sigma_2' = \sigma_2[\mathit{append}(\sigma_2.\rho_2.\mathrm{c},\ \sigma_2.\rho_2.\mathrm{d})/\rho_2.\mathrm{c},\ \mathit{false}/\rho_2.\mathrm{c.connect},\ \epsilon/\rho_2.\mathrm{d}]$

$\mathit{Isconnect}(\rho,\sigma) = (\exists \mathrm{c} \in \mathrm{ICI} : \sigma.\rho.\mathrm{c.connect} = \mathit{true})$

3.5.8 Parallel execute statement

To give the semantics such that each statements are executed concurrently, we apply the semantic function of each statement and then compose this result. Here, we apply the semantic function after the unused location l_0 is divided. It is to avoid the collision of the location of the local variable of the processes created in each statement. The composition of the result of the execution can be done as follows: $\mathit{Var}(S_1)$ is updated to the content of the state σ_1 , which is the result of executing the statement S_1. Similarly, $\mathit{Var}(S_2)$ is updated. If there is an input wait in either statement, to represent this, σ.wait is given the union of σ_i.wait of each statement S_i, where i is either 1 or 2. The unused location l_0 is the union of $\sigma_i.l_0$, which is the result of executing each statement.

$$\begin{aligned}\mathcal{C}[\![\ \mathrm{S_1}\ //\ \mathrm{S_2}\ ]\!]_g = (\ \lambda(\rho,\sigma):\ \{\ &(\rho,\ \sigma[\ \sigma_1.\rho.\mathit{Var}(\mathrm{S_1})\ /\rho.\mathit{Var}(\mathrm{S_1}),\ \sigma_2.\rho.\mathit{Var}(\mathrm{S_2})/\rho.\mathit{Var}(\mathrm{S_2}),\\ &\qquad (w_1 \wedge \sigma_1.\mathit{wait}) \vee (w_2 \wedge \sigma_2.\mathit{wait})/\mathit{wait},\ (\sigma_1.l_0 + \sigma_2.l_0)/l_0])\\ &\ |\ (\rho_1,\sigma_1) \in \mathcal{C}c[\![\mathrm{S_1}]\!]_g(\rho,\sigma[l_1/l_0]),\\ &\qquad (\rho_2,\sigma_2) \in \mathcal{C}c[\![\mathrm{S_2}]\!]_g(\rho,\sigma[l_2/l_0]),\ \mathit{partition}_2(\sigma.l_0) = (l_1,l_2)\ \}\ )\end{aligned}$$

where $w_i = \neg \mathit{SKIP}_\rho(\sigma, \mathit{LABEL}(\mathrm{S_i})),\ \ i = 1,2$

3.5.9 Nondeterministic select statement

If σ.wait is not true, the meaning is given the union of the result of each statement. Otherwise, we check in which statement an input wait is occured, then execute this statement.

$$\begin{aligned}\mathcal{C}[\![\ \mathrm{S}_1\ []\ \mathrm{S}_2\ ]\!]_g = (\lambda(\rho,\sigma) : \textbf{IF}\ & \sigma.wait = true \\ & \textbf{THEN}\ (\textbf{IF}\ \neg SKIP_\rho(\sigma, LABEL(\mathrm{S}_1)) \\ & \qquad \textbf{THEN}\ \mathcal{C}[\![\mathrm{S}_1]\!]_g(\rho,\sigma) \\ & \qquad \textbf{ELSE}\ \mathcal{C}[\![\mathrm{S}_2]\!]_g(\rho,\sigma)\ ) \\ & \textbf{ELSE}\ \mathcal{C}[\![\mathrm{S}_1]\!]_g(\rho,\sigma)\ \cup\ \mathcal{C}[\![\mathrm{S}_2]\!]_g(\rho,\sigma)\ )\end{aligned}$$

3.5.10 While statement

If σ.wait is false and the conditional expression is false, the function returns a set contained only a pair of original local environment and state, like an empty statement. Otherwise, we apply the semantic function of the statement following do, and then apply the result to the semantic function of the while statement again. But, if an input wait is occured when the statement following do is executed, the function returns only that result without recurring. The semantic function of while statement is the fix-point of this recursive function.

$$\begin{aligned}\mathcal{C}[\![\texttt{while}\ E\ \texttt{do}\ \mathrm{S}]\!]_g = fix(\lambda F : (\lambda(\rho,\sigma) : \mathtt{IF}\ & (\sigma.wait = false) \wedge (\mathcal{E}[\![E]\!]_\rho.\sigma = false) \\ & \textbf{THEN}\ \{\ (\rho,\sigma)\ \} \\ & \textbf{ELSE}\ (\mathcal{C}[\![\mathrm{S}]\!]_g \circ (\lambda(\rho_1,\sigma_1) : \textbf{IF}\ \sigma_1.wait = true \\ & \qquad \textbf{THEN}\ \{\ (\rho_1,\sigma_1)\ \} \\ & \qquad \textbf{ELSE}\ F(\rho_1,\sigma_1)\ )\ )(\rho,\sigma)\ ))\end{aligned}$$

3.5.11 Process call statement

Like a channel input statement, we assign a unique label to a process call statement. If σ.wait is false, we apply Proc_func, which is the result of the application the global environment to the process name, to the location of the actual argument and the current local environment. If σ.wait is true, we use the environment reserved in the field env of the process variable instead of the current local environment. Let the result state of the application be σ'. In both case, if σ'.wait is true, we record the lable to the field waitloc of the process variable and reserve the local environment which is the result of the application of the process function in the field env.

$$\begin{aligned}\mathcal{C}[\![L_i :\ \mathrm{p(a)}]\!]_g = (\lambda(\rho,\sigma) : \textbf{IF}\ & \sigma.wait = false \\ & \textbf{THEN}\ \{\ (\rho,\sigma'') \mid \exists(\rho',\sigma') \in (g(\sigma.\rho.\mathrm{p})).(\rho,\ \sigma).(\rho.\mathrm{a}), \\ & \qquad \sigma'' = (\textbf{IF}\ \sigma'.wait = false \\ & \qquad\qquad \textbf{THEN}\ \sigma[\sigma'.\rho.\mathrm{a}/\rho.\mathrm{a}] \\ & \qquad\qquad \textbf{ELSE}\ \sigma'[true/\rho.\mathrm{p.wait}, \\ & \qquad\qquad\qquad L_i/\rho.\mathrm{p.waitloc},\ \rho'/\rho.\mathrm{p.env}])\ \} \\ & \textbf{ELSE}\ \{\ (\rho,\sigma'') \mid \exists(\rho',\sigma') \in (g(\sigma.\rho.\mathrm{p})).(\sigma.\rho.\mathrm{p.env},\ \sigma).(\rho.\mathrm{a}), \\ & \qquad \sigma'' = (\textbf{IF}\ \sigma'.wait = false \\ & \qquad\qquad \textbf{THEN}\ \sigma[\sigma'.\rho.\mathrm{a}/\rho.\mathrm{a},\ false/wait,\ false/\rho.\mathrm{p.wait}] \\ & \qquad\qquad \textbf{ELSE}\ \sigma'[true/\rho.\mathrm{p.wait}, \\ & \qquad\qquad\qquad L_i/\rho.\mathrm{p.waitloc},\ \rho'/\rho.\mathrm{p.env}])\ \})\end{aligned}$$

4 Conclusion

In the denotational semantics for the parallel processing language proposed in this report , we suppress the operational aspect as possible as we can, then we can give a semantics to this language in the highly abstract level. Besides, the language presented in this report involves the fundamental facilties of parallel processing languages: parallel execution, communication between processes, nondeterministic selection and declaration of local variables. Then, adding data types and control structures, we can easily extend the language to a more practical language.

Acknowledgement

We thank Masakazu Nasu and Hiroshi Kamabe for helpful discussion.

References

[1] G.Kahn D.B, MacQueen: "Coroutines and network of parallel processes", IFIP 77 (1977) pp.993-998

[2] J.D.Brock W.B.Ackerman: "Scenarios:A Model of Nondeterminate Computation", LNCS 107 (1981) pp.252-259

[3] M.Broy C.Lengauer: "On Denotational versus Predicative Semantics", technical report of UNIVERSITY PASSAU Aug.(1987)

[4] G.D.Plotkin: "A Power Domain Construction", SIAM J.Compt. Sept.(1976)

[5] M.B.Smyth: "Power Domains", JCSS 16 (1978) pp.23-36

[6] J.E.Stoy: "Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory", MIT Press (1977)

A The memory of the location where an input wait is occured

An input channel variable c has four field (`c`,`c.waitloc`,`c.wait`, `c.connect`). On the state σ and the local environment ρ, these field have the following values.

$\sigma.\rho.c \in STREAM(D)$	the content of the channel
$\sigma.\rho.c.\texttt{waitloc} \in \texttt{id}$	the label where an input wait is occured
$\sigma.\rho.c.\texttt{wait} \in \{\bot, true, false\}$	the truth whether `waitloc` is valid or not
$\sigma.\rho.c.\texttt{connect} \in \{\bot, true, false\}$	the truth whether the channel is connected

A process variable p has four field (`p`,`p.waitloc`,`p.wait`,`p.env`). On the state σ and the local environment ρ, these field have the following values.

$\sigma.\rho.p \in \texttt{id}$	the name of the process
$\sigma.\rho.p.\texttt{waitloc} \in \texttt{id}$	the label where an input wait is occured
$\sigma.\rho.p.\texttt{wait} \in \{\bot, true, false\}$	the truth whether `waitloc` is valid or not
$\sigma.\rho.p.\texttt{env} \in \texttt{Lenv}$	the local environment of the process when an input wait is occured

B Auxiliary Functions

1. $LABEL(S)$
   The set consisting of the lables in the statement S.

2. $SKIP_\rho(\sigma, LABEL(S))$
   The predicate indicating that there is no wait in the statement S. That means the label memoried in the field `waitloc` of an input channel variable or a process variable is not contained in the $LABEL(S)$.

3. $Var(S)$
   All variables used in the statement S.

4. $getloc, removeloc, partition_n$
   These functions deals with locations. The function *getloc* extracts locations, the function *removeloc* removes used locations and the function $partition_n$ divides the set of locations into n partitions.

$$getloc : \mathcal{P}(\texttt{Location}) \to \texttt{Location}$$
$$removeloc : \mathcal{P}(\texttt{Location}) \times \texttt{Location} \to \mathcal{P}(\texttt{Location})$$
$$partition_n : \mathcal{P}(\texttt{Location}) \to (\mathcal{P}(\texttt{Location}))^n$$

C Stream

We can represent the data used with channels as the message sequence called "stream"($STREAM(D)$).

$$STREAM(D) = D^* \cup D^* \cdot \{\bot\} \cup D^\infty$$

We prepare the following functions for this stream.

$$first : STREAM(D) \to D^\bot$$
$$rest : STREAM(D) \to STREAM(D)$$
$$append : STREAM(D) \times STREAM(D) \to STREAM(D)$$

These functions are defined by the following equations. Where, $a \in D^\bot, s, s_1, s_2 \in STREAM(D)$.

$$first(\epsilon) = \bot$$
$$first(a \cdot s) = a$$
$$rest(\epsilon) = \bot$$
$$rest(a \cdot s) = \begin{cases} s & \text{if } a \neq \bot \\ \bot & \text{if } a = \bot \end{cases}$$
$$append(\epsilon \cdot s_2) = s_2$$
$$append((a \cdot s_1), s_2) = \begin{cases} a \cdot (append(s_1, s_2)) & \text{if } a \neq \bot \\ \bot & \text{if } a = \bot \end{cases}$$