Embedding of Moded Flat GHC into Polyadic π-Calculus

Keiji Hirata

NTT Basic Research Laboratories
3-1, Morinosato Wakamiya, Atsugi-shi,
Kanagawa, 243-01 Japan

hirata@nefertiti.ntt.jp

Abstract

This paper presents the embedding of Flat GHC into π-calculus and discusses their relationship. Although GHC and π-calculus share similar motivation and aims, they have quite different origins, concepts and formulations. Thus, their comparison may provide us with new insights into concurrent computation. The embedding is represented by the translation rules from an Flat GHC program to π-calculus statements. First, to make the translation simpler, FGHC$^-$ is introduced which is an appropriate subset of moded Flat GHC. Next, the translation rules from an FGHC$^-$ program to polyadic π-calculus statements are described. Then, as an example, the APPEND program in FGHC$^-$ is translated. Through the translation steps, the various features of Flat GHC, such as moded logical variables, passive and active unification, predicate invocation, concurrency and indeterminism, are reexamined in the terminology of π-calculus. Moreover, this embedding method is applied to specify built-in predicates. As an example, the specification of the built-in predicates to manipulate shared data is demonstrated. A translator has been implemented in KL1 which generates an object program, described in a programming language based on π-calculus, from an FGHC$^-$ program.

1 Introduction

Flat Guarded Horn Clauses (FGHC for short) is a concurrent logic language based on parallelism in logic programming, while π-calculus is a model of concurrent computation based on the notion of naming. Both are simple yet powerful frameworks for describing concurrent processes (referred to as *agents* in π-calculus).

A salient feature of GHC is that interprocess communication is achieved by unification, where logical variables play a crucial role, that is, dataflow synchronization and broadcasting. A program consisting of clauses can be viewed as a conditional rewrite rule of goals, and goal rewriting governs the unification of logical variables. The semantics of

GHC has been intensively studied and developed so far. Among the research on the relation between GHC and other similar paradigms, Honda et al. [4] compared GHC features with the Actor model, and described the implementation of GHC in actors.

During the computation of π-calculus, agents concurrently exchange names as data via names as channels, where a name is only an entity. A name enables us to encapsulate as a value an agent which can be instantiated, activated and applied, so we can treat it as a first-class object. At the same time, π-calculus provides an execution model such that independent states interact with one another. Its semantics has been investigated intensively mainly from the algebraic point of view. Moreover, connections to other similar models have also been elaborated: e.g. λ-calculus [6], object-oriented languages [14] and Prolog [5].

Although GHC and π-calculus share similar motivation and aims, they have quite different origins, concepts and formulations, as mentioned above. Investigations into the link between GHC and π-calculus have been few. Thus, their comparison may provide us with new insights into concurrent computation.

This paper discusses the connection by embedding FGHC into π-calculus. First, to make the translation simpler, we introduce FGHC^- which is a subset of moded FGHC. Next, the translation rules for an FGHC^- program to polyadic π-calculus (PPC) statements are described. This embedding illustrates how π-calculus represents the language features of FGHC. Lastly, the APPEND program in FGHC^- is translated to PPC statements as an example. Moreover, this embedding method is applied to specify built-in predicates, and the specification of the built-in predicates to manipulate shared data is actually demonstrated.

2 Preliminaries

2.1 Moded FGHC

This section briefly reviews moded FGHC [13], which is FGHC with mode information. In moded FGHC, every occurrence of a variable is associated with a mode, either *in* or *out*. A logical variable is considered a one-to-n ($n \geq 0$) communication channel that connects its occurrences. Roughly, an occurrence moded *in* and *out* corresponds to inlet and outlet of information, respectively. If you imagine that a variable is a MOSFET, *in* may remind you of a drain and *out* a source, although this analogy represents one-to-one communication. All variables in a well-moded program are guaranteed to be used for one-to-n ($n \geq 0$) communication; that is, there are one *out* occurrence and n *in* occurrences for each variable.

Some advantages of well-moded FGHC are informally stated below [13]:

- Given a well-moding of a program and a goal, if the goal is reduced by one step, the reduced goal and the program have the same well-moding.
- Given a well-moding of a program and a goal, if the goal has been reduced successfully to an empty set of goals, all the variables in the goal are mapped to ground terms.

The source language considered for the embedding is a subset of moded FGHC (FGHC^- for short), which has the following syntax:

$$
\begin{aligned}
Goal &::= \;:\!-\; b_1, \cdots, b_n \\
Program &::= c_1 \cdots c_n \\
c_i &::= u(\vec{A}) :\!- g \mid b_1, \cdots, b_n \\
g &::= true \mid A^i_j = f(\vec{D}) \mid A^i_j = A^i_k \mid g\_builtin(\vec{E}^i) \\
b_i &::= true \mid u(\vec{R}) \mid b\_builtin(\vec{R}) \mid Y^o_j = f(\vec{R}) \mid Y^o_j = X^i_k
\end{aligned}
$$

The elements of $var(\vec{A})$ and $var(\vec{D})$ are distinct.
$A^i_j, A^i_k \in \vec{A},\ var(\vec{A}) \cap var(\vec{D}) = \phi,\ \vec{E} \subseteq \vec{A}$

Let *Pred*, *Fun* and *Var* be the disjoint sets of predicate, function and variable symbols; $u \in Pred$ and $f \in Fun$. Here $\vec{V}$ means $V_1, \cdots, V_n$, $u(\vec{R})$ represents a user-defined predicate, and $var(\vec{V})$ means the set of variables in $\vec{V}$. Following convention, capital letters stand for logical variables. $g\_builtin(\vec{E}^i)$ and $b\_builtin(\vec{R})$ represents a guard built-in and a body built-in predicate, respectively. Every occurrence of a variable is associated with a mode, either *in* or *out* and is written V^i or V^o.

Compared to FGHC, FGHC$^-$ has some properties listed below:

- A goal and a program are well-moded.
- At the top level of arguments of the predicate and the function symbols, only variables occur.
- Function symbols occur only as the arguments of '=' both in the guard and the body.
- For a guard goal, only either a passive unification or a guard built-in predicate occurs.

It is obvious that FGHC$^-$ has almost the same descriptive power as FGHC, although FGHC$^-$ is syntactically restricted.

2.2 Communication Variable

The occurrence patterns of FGHC$^-$ variables are classified in Table 1. In the table, 'i (≥ 1)' means that there is more than one occurrence moded *in*.

Let a communication variable (*c-var* for short) be defined as a variable occurring with both *in* and *out* modes simultaneously in the body. Cases 3.1∼3.3 in Table 1 correspond to c-vars. Intuitively, in FGHC$^-$, the occurrence of a c-var implies that a variable which is single-assigned to and possibly multiple-referred to by other processes is allocated at that time. In particular, the c-var occurring in the body and moded *out* is written e.g. Y_j^c and $\vec{Z}^c$ in this paper. On the other hand, if a variable detected is to be used for a one-to-one communication channel, the variable is not a c-var. We do not care about whether or not the variables moded *in* is a c-var.

2.3 Polyadic π-Calculus

π-calculus agents (processes) exchange data through communication channels concurrently [7]. The data and the communication channels are represented by names, which are only entities in π-calculus. The syntax of polyadic π-calculus (PPC for short) is as follows:

Table 1: Occurrence Patterns of FGHC$^-$ Variables

	Head	Guard		Body
Case	$u(\vec{A})$	A^i_j	$f(\vec{D})$	$b_1, \cdots, b_n$
1.1	i			$i\ (\geq 1)$
1.2	i	i		$i\ (\geq 1)$
1.3			o	$i\ (\geq 1)$
2.1	o			o
2.2			i	o
3.1				o and $i\ (\geq 1)$
3.2	o			o and $i\ (\geq 1)$
3.3			i	o and $i\ (\geq 1)$

$$P ::= \textstyle\sum_{i \in I} \pi_i.P_i \;\Big|\; P|Q \;\Big|\; (\nu x)P \;\Big|\; \mathcal{A}(x) \;\Big|\; !P \;\Big|\; \mathbf{0}$$

π_i is called prefix and has two basic forms: $\overline{x}[y_1, \cdots, y_n]$ (polyadic output) and $x([y_1, \cdots, y_n])$ (polyadic input). $\overline{x}[y_1, \cdots, y_n].P$ sends the tuple $[y_1, \cdots, y_n]$ along the name x as an atomic action and then behaves as P. On the other hand, $x([y_1, \cdots, y_n]).P$ atomically receives a tuple $[z_1, \cdots, z_n]$ along the name x and behaves as $P\{z_1/y_1, \cdots, z_n/y_n\}$. Thus, in a sense, $x([y_1, \cdots, y_n])$ works like the lambda prefix $\lambda y_1 \cdots y_n$. $P \mid Q$ represents concurrent agents P and Q interacting through names that they share. $(\nu x)P$ means that x is private to P; this is interpreted as a fresh name declaration which is local to P as well. And $\mathbf{0}$ means an inactive agent; this term is no longer reduced. We often write α as an abbreviation for $\alpha.\mathbf{0}$.

The following examples show how reduction proceeds (supposing that x does not occur freely in P and Q):

$$\begin{aligned}
(\nu xy)(\overline{x}y \mid x(z).P) &\longrightarrow (\nu y)(P\{y/z\}) \\
(\nu xyz)(\overline{x}y \mid x(z).P \mid z(w).Q) &\longrightarrow (\nu yz)(P\{y/z\} \mid z(w).Q) \\
(\nu xy)(\overline{x}y \mid y(x)) &\longrightarrow \mathit{irreducible},
\end{aligned}$$

where $\longrightarrow$ means one-step reduction at the PPC level.

$\mathcal{A}$ is an agent identifier that is associated with the agent definition $\mathcal{A}(x) \stackrel{\text{def}}{=} P$. The agent identifier is regarded as a lazy macro expansion from the execution point of view. The following is an example of an agent definition:

$$\begin{aligned}
&f[p] \stackrel{\text{def}}{=} p(x).(\overline{x}() \mid f[p]) \\
&(\nu abp)(f[p] \mid \overline{p}a \mid a() \mid \overline{p}b \mid b() \;\;\longrightarrow^* \;\; (\nu bp)(f[p] \mid \overline{p}b \mid b()) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \longrightarrow^* \;\; \mathbf{0}
\end{aligned}$$

To end, we use $\stackrel{\text{def}}{=}$ for the definition of an agent including recursive calls; while we use macro expansion, indicated as $=$, if an agent does not include recursive calls.

A summation $\sum_{i \in I} \pi_i.P_i$ behaves like one or other of the P_i. Although ! represents replication, our embedding does not use ! since it is difficult to practically deal. Instead, a guarded recursive call is used.

3 Embedding

This section describes how to translate a source program in FGHC$^-$ to agent definitions in PPC.

3.1 Representation of FGHC Terms in Polyadic π-Calculus

We start with Milner's [7] idea for representing general data structures. Let us consider the following sorting:

$$\{ \ TERM \mapsto (CONS, NIL, INT), \\ CONS \mapsto (TERM, TERM), \ NIL \mapsto (), \ INT \mapsto (V) \ \}$$

According to the sorting, the macro definitions for *cons*, *nil* and *int* are given by:

$$\begin{array}{rcl} cons[t,x,y] & = & t([c,n,i]).\overline{c}[x,y] \\ nil[t] & = & t([c,n,i]).\overline{n}() \\ int[t,v] & = & t([c,n,i]).\overline{i}[v] \end{array}$$

For instance, we may assume that the *cons* definition represents the following protocol sequence: (1) a *cons* structure is located at t, (2) a reader agent sends a tuple $[c,n,i]$ as a probe via t, (3) after receiving the tuple the name c is picked up, and (4) the tuple $[x,y]$ is returned to the reader via c. Since c is used for a channel, the reader can see that the sort is *cons* and the return value is a two-element tuple. This method can encode a function symbol with any arity. Let a PPC statement in the above form be called *c-agent* and $[c,n,i]$ *sort tuple*. A sort tuple should include all the sorts occurring in an FGHC$^-$ program to be translated.

The following PPC statement reads out the value of a c-agent:

$$(\nu p)((\nu xy)(p([c,n,i]).\overline{c}[x,y]) \mid \underline{(\nu cni)(\overline{p}([c,n,i]) \mid c([a,b]).P)}) \\ \longrightarrow^{*} \ (\nu xy)(P\{x/a, y/b\})$$

Let the underlined fragment be called *a-agent* [1]. Once an a-agent reads a c-agent, the c-agent terminates.

For simplicity, we introduce abbreviations for a c-agent and an a-agent:

$$\begin{array}{lrcl} \text{c-agent:} & p\langle\overline{s}\rangle[\vec{v}] & \equiv & p([\mathsf{st}]).\overline{s}[\vec{v}] \\ \text{a-agent:} & \overline{p}\langle s\rangle(\lambda\vec{v})Q & \equiv & \overline{p}[\mathsf{st}] \mid s([\vec{v}]).Q \end{array}$$

where p means the location of the agents, s the sort and $\vec{v}$ the values, and st stands for a sort tuple. This notation avoids having to explicitly write sort tuples. Now let us consider the reduction of a c-agent and an a-agent, again:

$$(\nu p)(\overline{p}\langle s\rangle(\lambda\vec{v})Q \mid p\langle\overline{s}\rangle[\vec{w}]) \longrightarrow^{*} Q\{\vec{w}/\vec{v}\}$$

Our embedding scheme interprets this reduction at the PPC level as follows; (1) the structure $s(\vec{w})$ is actively unified with a variable, (2) the variable is then passively unified

[1] C for c-agent and a for a-agent originates from concretion and abstraction, respectively, according to [7].

with $s(\vec{v})$, (3) the substitution $\{\vec{w}/\vec{v}\}$ is consequently obtained, and (4) the execution of $Q\{\vec{w}/\vec{v}\}$ proceeds. Interestingly, our embedding also uses the form of this reduction to achieve predicate invocation in FGHCpm^{-}.

Next, we introduce a forwarder agent:

$$\mathsf{fw}[r,w] \;=\; r(t).\overline{w}t$$

The forwarder represents *dereference* as follows:

$$(\nu pq)(\overline{p}\langle s\rangle(\lambda\vec{v})Q \mid \mathsf{fw}[p,q] \mid q\langle\vec{s}\rangle[\vec{w}]) \longrightarrow^{\star} Q\{\vec{w}/\vec{v}\}$$

$$(\nu pqr)(\overline{p}\langle s\rangle(\lambda\vec{v})Q \mid \mathsf{fw}[p,q] \mid \mathsf{fw}[q,r] \mid r\langle\vec{s}\rangle[\vec{w}]) \longrightarrow^{\star} Q\{\vec{w}/\vec{v}\}$$

However, if there is no a-agent, the reduction cannot proceed:

$$(\nu pqr)(\mathsf{fw}[p,q] \mid \mathsf{fw}[q,r] \mid r\langle\vec{s}\rangle[\vec{w}]) \longrightarrow \textit{irreducible}$$

The forwarder can connect a c-agent and an a-agent. To connect a c-agent and $n(\geq 2)$ a-agents, we introduce a new PPC agent, $\mathsf{var}[r,w]$. As mentioned in Section 2.1, a variable occurrence of moded FGHC has a mode, *in* or *out*. The *in* and the *out* occurrence corresponds to r and w for $\mathsf{var}[r,w]$, respectively, since the w channel works as an outlet of data and the r channel inlets (Fig. 1). The agent $\mathsf{var}[r,w]$ can synchronize write and

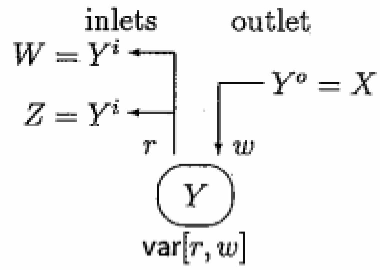

Figure 1: Moded Logical Variable and $\mathsf{var}[r,w]$

read operations and duplicate the data written via the w channel as many times as read operations are issued. Here $\mathsf{var}[r,w]$ and its auxiliary agents are defined as:

$$\begin{aligned}
\mathsf{var}[r,w] = \;& (\nu cn)(\overline{w}[c,n,i] \mid (c([x,y]).\mathsf{rep_cons}[r,x,y] \\
& \qquad\qquad + n().\mathsf{rep_nil}[r] \\
& \qquad\qquad + i([v].\mathsf{rep_int}[r,v])) \\
\mathsf{rep_cons}[p,x,y] \stackrel{\text{def}}{=}\;& p([c,n,i]).(\overline{c}[x,y] \mid \mathsf{rep_cons}[p,x,y]) \\
\mathsf{rep_nil}[p] \stackrel{\text{def}}{=}\;& p([c,n,i]).(\overline{n}() \mid \mathsf{rep_nil}[p]) \\
\mathsf{rep_int}[p,v] \stackrel{\text{def}}{=}\;& p([c,n,i]).(\overline{i}[v] \mid \mathsf{rep_int}[p,v])
\end{aligned}$$

The sort tuple is $[c,n,i]$ (implying $cons/2$, $nil/0$, $int/1$, respectively). The agents whose symbol names begin with rep_ are responsible for data duplication. On the other hand, $\mathsf{fw}[r,w]$ just synchronizes write and read operations and does not duplicate data.

3.2 Translation Rules

The notation $[\![t]\!]p$ represents a PPC agent to which the FGHC^{-} term t is translated and which includes the name p added by the translator.

T1: Clauses An entire program in FGHC$^-$ is translated to an agent definition, $clause[p]$, in PPC. That is, given a set of all clauses $c_1 \cdots c_m$ of all predicates,

$$clause[p] \stackrel{def}{=} (\nu cr)(\mathsf{var}[r,p] \mid \overline{c}() \mid [\![c_1]\!]rc \mid \cdots \mid [\![c_m]\!]rc)$$

is created. The definition clauses for different predicates are merged in an agent definition. Here, $\mathsf{var}[r,p]$ duplicates a predicate invocation as many times as the number of definition clauses.

T2: Head For this translation, the vocabulary with which source programs are written in FGHC$^-$ is used as the sort names of PPC as it is. However, for distinction the FGHC$^-$ variables are described in capital letters, while the PPC names are in small letters; e.g., the variables of FGHC$^-$, $\vec{A}$, correspond to the sort names of PPC, $\vec{a}$. Each clause c_i is translated to $[\![c_i]\!]rc$. The term $\vec{b}$ stands for $b_1, \cdots, b_n$.

$$\begin{aligned} [\![u(\vec{A}) :- g \mid \vec{b}]\!]rc &= (\nu \mathsf{st})(\overline{r}[\mathsf{st}] \mid u([\vec{a}]).[\![g \mid \vec{b}]\!]c) \\ &\equiv \overline{r}\langle u\rangle(\lambda\vec{a})[\![g \mid \vec{b}]\!]c \end{aligned}$$

The translation of a head literal works as an a-agent. Note that a commit operator of FGHC$^-$ is distinguished from parallel composition of PPC, although both are indicated by '|'.

T3: Guard The translation, $[\![g \mid \vec{b}]\!]c$, is given by:

$$\begin{aligned} [\![true]\!]c &= c().[\![\vec{b}]\!] \\ [\![A^i_j = f(\vec{D}) \mid \vec{b}]\!]c &= (\nu \mathsf{st})(\overline{a_j}[\mathsf{st}] \mid f([\vec{d}]).c().[\![\vec{b}]\!]) \\ &\equiv \overline{a_j}\langle f\rangle(\lambda\vec{d})(c().[\![\vec{b}]\!]) \\ [\![A^i_j = A^i_k \mid \vec{b}]\!]c &= (\nu s)(\mathsf{p\_unif}[s,a_j,a_k] \mid s().c().[\![\vec{b}]\!]) \\ [\![g\_builtin(\vec{E}^i) \mid \vec{b}]\!]c &= (\nu s)(\mathsf{g\_builtin}[s,\vec{e}] \mid s().c().[\![\vec{b}]\!]) \end{aligned}$$

The term decomposition by the passive unification, $A^i_j = f(\vec{D})$, is implemented by an a-agent; then, the obtained substitution is applied to $[\![\vec{b}]\!]$. The passive unification of two variables is translated to a system-supported agent $\mathsf{p\_unif}[s,a,b]$ (details in Section 3.3). Here, $\mathsf{p\_unif}[s,a,b]$ reads in the information of instantiation via the a and b channels that correspond to the two logical variables. If a and b are instantiated to the same ground structure, $\mathsf{p\_unif}[s,a,b]$ sends a signal via s. Thus, the passive unification of our embedding conforms to that of KL1 [12].

The prefix $c()$ implements a commit operator. Since all the clauses of all predicates are merged into an agent $clause[p]$, in general, more than one $c()$ waits for a signal from $\overline{c}()$ introduced in T1. Hence, from the property of π-calculus reduction, with respect to indeterministic clause selection, a translated agent behaves as follows:

- If an instantiation is enough to achieve guard checking of a clause, the clause is selected.
- If more than one clause can be selected, any one of them will eventually be selected.
- Clauses that cannot be selected never produce a side effect (never export instantiation).

T4: Body Goals Translation of the goal and the body, $b_1, \cdots, b_n$ is given by:

$$[\![\vec{b}]\!] = (\nu\vec{w})((\nu p_1)([\![b_1]\!]p_1 \mid clause[p_1]) \mid \cdots \mid (\nu p_m)([\![b_m]\!]p_m \mid clause[p_m]) \mid [\![b_{m+1}]\!] \mid \cdots \mid [\![b_n]\!])$$

Let $\vec{W}$ be a set of c-vars moded *out* except for the variables occurring either in the head or in the guard (case 3.1 in Table 1). The above $\vec{w}$ corresponds to the $\vec{W}$. And $b_1 \cdots b_m$ $(0 \leq m \leq n)$ are user-defined predicates. The c-agent $[\![b_i]\!]p_i$ $(0 \leq i \leq m)$ and the a-agent $clause[p]$ defined in T1 form a predicate invocation.

T5: Predicate The arguments of predicates are divided into c-vars moded *out* $(\vec{Z}^c)$ and the rest $(\vec{R})$. The translation is given by:

$$\begin{aligned}
[\![true]\!] &= \mathbf{0} \\
[\![u(\vec{R}, \vec{Z}^c)]\!]p &= (\nu v_1 \cdots v_l)(p([\mathsf{st}]).\overline{u}[\vec{r}, v_1, \cdots v_l] \mid \mathsf{var}[z_1, v_1] \mid \cdots \mid \mathsf{var}[z_l, v_l]) \\
&\equiv (\nu v_1 \cdots v_l)(p\langle\overline{u}\rangle[\vec{r}, v_1, \cdots, v_l] \mid \mathsf{var}[z_1, v_1] \mid \cdots \mid \mathsf{var}[z_l, v_l]) \\
[\![b_builtin(\vec{R}, \vec{Z}^c)]\!] &= (\nu v_1 \cdots v_l)(\mathsf{b_builtin}[\vec{r}, v_1, \cdots v_l] \mid \mathsf{var}[z_1, v_1] \mid \cdots \mid \mathsf{var}[z_l, v_l])
\end{aligned}$$

Since the occurrence of a c-var moded *out* acts as a source for one-to-n communication, $\mathsf{var}[r, w]$ should duplicate the data on demand that is supplied just once by the source. A predicate is obviously implemented by a c-agent.

T6: Active Unification We need to provide two kinds of translation rules for a body goal b_i for the cases including and not including c-var's.

$$\begin{aligned}
[\![Y_i^o = f(\vec{R}, \vec{Z}^c)]\!] &= [\![f(\vec{R}, \vec{Z}^c)]\!]y_i \\
[\![Y_i^c = f(\vec{R}, \vec{Z}^c)]\!] &= (\nu v)(\mathsf{var}[y_i, v] \mid [\![f(\vec{R}, \vec{Z}^c)]\!]v) \\
[\![Y_j^o = X_k^i]\!] &= \mathsf{fw}[y_j, x_k] \\
[\![Y_j^c = X_k^i]\!] &= \mathsf{var}[y_j, x_k] \\
[\![f(\vec{R}, \vec{Z}^c)]\!]p &= (\nu v_1 \cdots v_l)(p([\mathsf{st}]).\overline{f}[\vec{r}, v_1, \cdots, v_l] \mid \mathsf{var}[z_1, v_1] \mid \cdots \mid \mathsf{var}[z_l, v_l]) \\
&\equiv (\nu v_1 \cdots v_l)(p\langle\overline{f}\rangle[\vec{r}, v_1, \cdots, v_l] \mid \mathsf{var}[z_1, v_1] \mid \cdots \mid \mathsf{var}[z_l, v_l])
\end{aligned}$$

Note that the translation method for a function symbol $f(\vec{R}, \vec{Z}^c)$ is the same as that for a user-defined predicate $u(\vec{R}, \vec{Z}^c)$. Thus, considering T1 through T6 together, both the invocation of a user-defined predicate and the passive and active unification are achived by the same mechanism, that is, the combination of a c-agent and a-agents.

3.3 System-Supported PPC Agents

The translator must provide a runtime library that reflects the information of all sort names occurring in a given source program. The runtime library consists of $\mathsf{p\_unif}[s, a, b]$ for the passive unification of two variables, guard and body built-in predicates, $\mathsf{var}[r, w]$ and $\mathsf{fw}[r, w]$.

Supposing a sort tuple is $[c, n, i]$ (implying $cons/2$, $nil/0$, $int/1$, respectively), $\mathsf{p\_unif}[s, a, b]$ is defined as:

$$
\begin{aligned}
\mathsf{p\_unif}[s,a,b] \stackrel{\text{def}}{=} \; & (\nu c_a n_a i_a c_b n_b i_b)(\overline{a}[c_a,n_a,i_a] \mid \overline{b}[c_b,n_b,i_b] \mid \\
& \quad ((c_a([x_a,y_a]).c_b([x_b,y_b]).(\nu s_x s_y)(\mathsf{p\_unif}[s_x,x_a,x_b] \\
& \qquad \mid \mathsf{p\_unif}[s_y,y_a,y_b] \\
& \qquad \mid s_x().s_y().\overline{s}())) \\
& \quad + (n_a().n_b().\overline{s}()) \\
& \quad + (i_a([v_a]).i_b([v_b]).\mathsf{p\_unif}[s,v_a,v_b])))
\end{aligned}
$$

The guard built-in predicates, such as $X \leq 0$, $X > Y$ and $print(X)$, are translated to the agents of the form $\mathsf{g\_builtin}[s,\vec{e}]$; $\vec{e}$ represents input arguments and s is a channel via which the guard built-in agent sends a signal for successful termination. The body built-in predicates, such as $modulo(X,Y,M)$ and $minus1(X,R)$, are of the form $\mathsf{b\_builtin}[\vec{r}]$. While this agent does not have a signal channel, $\vec{r}$ possibly includes input and output arguments. The definitions of $\mathsf{var}[r,w]$ and $\mathsf{fw}[r,w]$ are already given in Section 3.1, which supposes the same sort tuple, $[c,n,i]$.

As mentioned earlier, a well-moded FGHC program is guaranteed to have the single-write property. Thus, even if all $+$'s are substituted by $|$'s in the above definition, $\mathsf{p\_unif}[s,a,b]$ (and $\mathsf{var}[r,w]$ as well) can work, since only one of its sub-agents is activated.

4 Example: APPEND Program

This section demonstrates the translation of the APPEND program in FGHC$^-$. The source program is:

$$
\begin{aligned}
&app(X^i,Y^i,Z^o) \;\text{:-}\; X^i = nil && \mid Y^i = Z^o. \\
&app(X^i,Y^i,Z^o) \;\text{:-}\; X^i = cons(E^m,X^o_s) && \mid Z^o = cons(E^{\overline{m}},Z^i_s), app(X^i_s,Y^i,Z^o_s).
\end{aligned}
$$

Z_s in the second clause is a c-var, and its second occurrence should be treated carefully (i.e. $Z_s{}^c$). The translation result is shown below:

$$
\begin{aligned}
clause[v_0] \stackrel{\text{def}}{=} \; & (\nu v_1 v_2)(var[v_2,v_0] \\
& \mid \overline{v_1}() \\
& \mid (\nu \mathsf{st})(\overline{v_2}[\mathsf{st}] \mid app([x,y,z]).(\nu \mathsf{st})(\overline{x}[\mathsf{st}] \mid nil().v_1().\mathsf{fw}[z,y])) \\
& \mid (\nu \mathsf{st})(\overline{v_2}[\mathsf{st}] \mid app([x,y,z]).(\nu \mathsf{st})(\overline{x}[\mathsf{st}] \mid cons([e,x_s]).v_1(). \\
& \qquad (\nu z_s)((\nu v_{17})((\nu v_{18})(v_{17}([\mathsf{st}]).\overline{app}[x_s,y,v_{18}] \\
& \qquad\qquad \mid \mathsf{var}[z_s,v_{18}]) \\
& \qquad\quad \mid clause[v_{17}]) \\
& \qquad \mid z([\mathsf{st}]).\overline{cons}[e,z_s])))))
\end{aligned}
$$

In the above agent definition, st stands for a sort tuple and includes $cons$, nil, app. The names v_n are created during the translation; e.g., v_1 implements a commit operator. Although the agent identifiers, $\mathsf{var}[r,w]$ and $\mathsf{fw}[r,w]$, should be macro-expanded in place, the result without macro expansion is shown above for readability of the translated PPC code. The two clauses in FGHC$^-$ are merged into one agent definition. The active unification in the first clause $Y^i = Z^o$ is translated to $\mathsf{fw}[z,y]$. The c-var Z_s in the second clause is translated to $\mathsf{var}[z_s,v_{18}]$. In addition, the translator also must generate the proper runtime library.

5 Specification of Built-in Predicates

5.1 New Built-in Predicate for Shared Data Manipulation

This section specifies a new built-in predicate of FGHC^{-} for shared data manipulation in PPC [2]. Let us call the predicate *swap_shared_data*(P, Old, New, Q), which is abbreviated to *swap_sd*, here. Figure 2 shows its operation. The arguments P and Q represent the

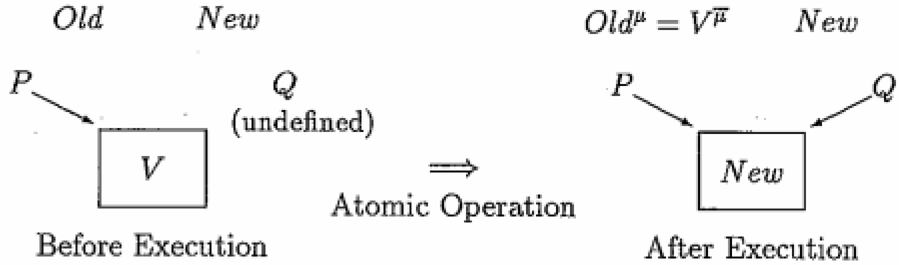

Figure 2: Operation of *swap_shared_data*/4

reference to the shared data (the location where the shared data exists). Suppose that the location P holds the value V; then, if *swap_sd*/4 is carried out, V referred to by P is replaced by New, and Old is unified with V. On the completion of the swap operation, Q is lastly unified with the reference to the shared data. This sequence of operations is performed atomically. It is different from *set_vector_element*/5 of KL1 [12] in that no duplication occurs even if P is shared by other processes; that is, data are always destructively rewritten in place.

By using *swap_sd*/4, we can implement a nondeterministic merger without using non-determinism among clauses which FGHC has by nature. The program is given by:

$$
\begin{array}{lll}
merge(X^i, Y^i, Z^o) \text{ :- } true & | & alloc\_sd(P^o), \\
 & & swap\_sd(P^i, I^i, Z^o, Q^o),\ I^o = init, \\
 & & mg\_in(X^i, Q^i),\ mg\_in(Y^i, Q^i).
\end{array}
$$

$$
\begin{array}{lll}
mg\_in(X^i, \_^i) \text{ :- } X^i = nil & | & true. \\
mg\_in(X^i, P^i) \text{ :- } X^i = cons(H^m, T^o) & | & swap\_sd(P^i, A^i, R^o, Q^o), A^o = cons(H^{\overline{m}}, R^i), \\
 & & mg\_in(T^i, Q^i).
\end{array}
$$

Depending on the context where the merger is used, mode m is determined either *in* or *out*. Next, the shared data manipulator *swap_sd*/4 and the shared data allocator *alloc_sd*/1 are specified in PPC below:

$$
\begin{array}{rcl}
swap\_sd[p, o, n, q] & = & \overline{p}\langle s\rangle(\lambda m)(\nu r)(\overline{m}[r, n].(r(v).\mathrm{fw}[v, o] \mid q\langle\overline{s}\rangle[m]))) \\
alloc\_sd[p] & = & (\nu m)(p\langle\overline{s}\rangle[m] \mid (\nu v)(swap[m, v])) \\
swap[m, v] & \stackrel{\mathrm{def}}{=} & m([r, n]).(\overline{r}v \mid swap[m, n])
\end{array}
$$

A new sort s should be added to the sort tuple, correspoinding to the introduction of a new data structure allowing destructive rewrite. Note that $\overline{m}[r, n]$ in the definition of *swap_sd* guarantees the atomicity of the swap operation. The modes of the *swap_sd*/4's

[2] The basic idea of this predicate originated from Mr. K. Kumon (ICOT).

arguments in the above *merge* definition are $swap\_sd(P^i, O^i, N^o, Q^o)$. To implement $swap\_sd(P^i, O^o, N^i, Q^o)$, the invocation $\mathsf{fw}[v, o]$ in the *swap_sd* definition should be replaced by $\mathsf{fw}[o, v]$.

Now, we can obtain the translated PPC statements for our merger show below:

$$
\begin{aligned}
clause[v_0] \stackrel{\text{def}}{=} (\nu v_1 v_2)(&\mathsf{var}[v_2, v_0] \\
&| \ \overline{v_1}() \\
&| \ (\nu \mathsf{st})(\overline{v_2}[\mathsf{st}] \mid merge([x, y, z]).v_1(). \\
&\qquad\qquad (\nu piq)((\nu v_{17})(v_{17}([\mathsf{st}]).\overline{mg\_in}[x, q] \mid clause[v_{17}]) \\
&\qquad\qquad\quad | \ (\nu v_{18})(v_{18}([\mathsf{st}]).\overline{mg\_in}[y, q] \mid clause[v_{18}]) \\
&\qquad\qquad\quad | \ (\nu v_{19})(alloc\_sd[v_{19}] \mid \mathsf{var}[p, v_{19}]) \\
&\qquad\qquad\quad | \ (\nu v_{20})(swap\_sd[p, i, z, v_{20}] \mid \mathsf{var}[q, v_{20}]) \\
&\qquad\qquad\quad | \ (\nu v_{21})(\mathsf{var}[i, v_{21}] \mid v_{21}([\mathsf{st}]).\overline{init}()))) \\
&| \ (\nu \mathsf{st})(\overline{v_2}[\mathsf{st}] \mid mg\_in([x, p]).(\nu \mathsf{st})(\overline{x}[\mathsf{st}] \mid nil().v_1())) \\
&| \ (\nu \mathsf{st})(\overline{v_2}[\mathsf{st}] \mid mg\_in([x, p]).(\nu \mathsf{st})(\overline{x}[\mathsf{st}] \mid cons([h, t]).v_1(). \\
&\qquad\qquad (\nu arq)((\nu v_{22})(v_{22}([\mathsf{st}]).\overline{mg\_in}[t, q] \mid clause[v_{22}]) \\
&\qquad\qquad\quad | \ (\nu v_{23} v_{24})(swap\_sd[p, a, v_{23}, v_{24}] \mid \mathsf{var}[r, v_{23}] \mid \mathsf{var}[q, v_{24}]) \\
&\qquad\qquad\quad | \ (\nu v_{25})(\mathsf{var}[a, v_{25}] \mid v_{25}([\mathsf{st}]).\overline{cons}[h, r])))))
\end{aligned}
$$

The new sort s in the definitions of *swap_sd* and *alloc_sd* always sends and receives the identical name m, which has been declared during initialization in *alloc_sd* (the data passing through the channel v_{19} in the above definition). Since *swap_sd*/4 and *alloc_sd*/1 are built-in predicates, they are translated into the invocation of PPC agents with the same names, according to T5. The PPC statements of our merger can be actually reduced at the PPC level. From the result of this reduction, we can verify that *merge*/3 defined in FGHC$^-$ can work like the conventional merger without using *swap_sd*. However, taking into account the case in which streams are closed by putting *nil*, *merge*/3 defined here is not identical. There are two ways to make *merge*/3 identical to the conventional merger; record the number of input streams by a counter implemented by *swap_sd*/4, or detect the termination of all the *mg_in*/2 processes by the short circuit method.

5.2 Specification of Port

Besides *swap_sd*/4, many primitives for shared data manipulation in concurrent logic languages have been proposed, such as port [2] and mutual reference cell [10] (MRC). To uniformly describe them in PPC makes fair comparison possible.

First, let us consider port. The *open_port*/2 primitive creates a port, and the *send*/2 primitive sends a data structure to the port. An example of their usage is given by:

$$:\!-\ open\_port(P, S),\ send(P, 2),\ send(P, 3),\ send(P, 5), \cdots.$$

Here, S is nondeterministically instantiated to either $[2, 3, 5, \cdots]$, $[3, 2, 5, \cdots]$, $[3, 5, 2, \cdots]$ $\cdots$. Suppose that the port primitives can be defined in FGHC$^-$ by *alloc_sd*/1 and *swap_sd*/4 as follows:

$$
\begin{aligned}
open\_port(P^o, Z^o) &\ :\!-\ true \mid alloc\_sd(S^o), swap\_sd(S^i, I^i, Z^o, P^o), I^o = init. \\
send(P^i, E^m) &\ :\!-\ true \mid swap\_sd(P^i, A^i, B^o, \_^o), A^o = cons(E^m, B^i).
\end{aligned}
$$

To verify these definitions, they are firstly translated into PPC and, then, are slightly simplified (reduced) at the PPC level. So, we obtain their PPC definitions given by [3]:

$$\begin{aligned} \mathit{open\_port}[p,z] &= (\nu m)(p\langle\overline{s}\rangle[m] \mid \mathit{swap}[m,z]) \\ \mathit{send}[p,e] &= (\nu a_w a_r b_w b_r v)(\mathit{swap\_sd}[p,a_r,b_w,v] \\ &\qquad \mid \mathsf{var}[a_r,a_w] \mid a_w\langle\overline{\mathit{cons}}\rangle[e,b_r] \mid \mathsf{var}[b_r,b_w]) \end{aligned}$$

Now, we translate a simple FGHC$^-$ goal,

$$:\!-\ \mathit{open\_port}(P,Z), \mathit{send}(P,E_1), \mathit{send}(P,E_2).$$

and examine the result of reduction. The above goal is translated into the following PPC statement, $(\nu p_w p_r)(\mathit{open\_port}[p_w,z] \mid \mathsf{var}[p_r,p_w] \mid \mathit{send}[p_r,e_1] \mid \mathit{send}[p_r,e_2])$. Since this statement can be reduced nondeterministically, an example of its reduction is demonstrated below:

$$\begin{aligned} &(\nu p_w p_r)(\mathit{open\_port}[p_w,z] \mid \mathsf{var}[p_r,p_w] \mid \mathit{send}[p_r,e_1] \mid \mathit{send}[p_r,e_2]) \\ &\longrightarrow^* (\nu p_r p_w)((\nu m)(p_w\langle\overline{s}\rangle[m] \mid \mathit{swap}[m,z]) \mid \mathsf{var}[p_r,p_w] \\ &\qquad \mid (\nu a_r a_w b_r b_w v)(\mathit{swap\_sd}[p_r,a_r,b_w,v] \mid \mathsf{var}[a_r,a_w] \\ &\qquad\qquad \mid a_w\langle\overline{\mathit{cons}}\rangle[e_1,b_r] \mid \mathsf{var}[b_r,b_w]) \\ &\qquad \mid \mathit{send}[p_r,e_2]) \\ &\longrightarrow^* (\nu p_r)((\nu b_w)((\nu m)(\mathit{swap}[m,b_w] \mid \mathsf{rep\_sd}[p_r,m]) \\ &\qquad \mid (\nu a_r b_r)(\mathsf{fw}[z,a_r] \mid \mathsf{rep\_cons}[a_r,e_1,b_r] \mid \mathsf{var}[b_r,b_w])) \\ &\qquad \mid \mathit{send}[p_r,e_2]) \\ &\longrightarrow^* (\nu p_r)((\nu b_w d_w)((\nu m)(\mathit{swap}[m,d_w] \mid \mathsf{rep\_sd}[p_r,m]) \\ &\qquad \mid (\nu a_r b_r)(\mathsf{fw}[z,a_r] \mid \mathsf{rep\_cons}[a_r,e_1,b_r] \mid \mathsf{var}[b_r,b_w]) \\ &\qquad \mid (\nu c_r d_r)(\mathsf{fw}[b_w,c_r] \mid \mathsf{rep\_cons}[c_r,e_2,d_r] \mid \mathsf{var}[d_r,d_w]))) \\ &\longrightarrow^* (\nu p_r)((\nu d_w)((\nu m)(\mathit{swap}[m,d_w] \mid \mathsf{rep\_sd}[p_r,m]) \\ &\qquad \mid (\nu a_r b_r d_r)(\mathsf{fw}[z,a_r] \mid \mathsf{rep\_cons}[a_r,e_1,b_r] \mid \mathsf{rep\_cons}[b_r,e_2,d_r] \\ &\qquad \mid \mathsf{var}[d_r,d_w]))) \end{aligned}$$

The above $\mathsf{rep\_sd}[p_r,m]$ processes, invoked within $\mathsf{var}[p_r,p_w]$, duplicate the value m. This reduction tells us that the *cons* chain, including e_1 and e_2 as its elements, is being constructed at the location z.

Also, we can proceed with the specification and verification of MRC in the same way. The creation and the access primitives for MRC are *allocate_mutual_reference*/2 and *stream_append*/3, respectively. As a result, they are also specified in PPC, although the PPC specifications are not presented here for space limitation. When you look at the specifications, it is clear that the specification of *open_port*/2 and that of *allocate_mutual_reference*/2 are identical and the specification of *stream_append*/2 is quite similar to that of *send*/2.

6 Related Work

The SCL Language: The purpose of SCL (simple concurrent language) [8] was to examine the various properties of concurrent logic languages from a semantically simpler

[3]According to the full port definition, when a system detects that there are no references to a port, the system automatically closes the port. However, this port termination is not taken into account here for simplicity.

and clearer standpoint. Actually, SCL was designed without the notion of a logical variable. Nyström showed the translation of FGHC to SCL. Although the term formation in SCL is quite different from ours presented in this paper, the basic idea for the execution of FGHC programs is similar. Thus, SCL cannot either treat the passive unification of non-ground terms or the failure of occur check.

Specification of Prolog: Li [5] translated the Prolog's control strategy to (non-polyadic) π-calculus, and then encoded the Prolog terms using the disjoint union type. Therefore, some primitives for equality checking and type checking at the π-calculus level are needed. Although this might be an obstacle for formal treatment, Li successfully encoded a non-moded logical variable.

γ-calculus and Oz: To express object-oriented features and higher-order functions, π-calculus has been extended by adding a logical variable [11] [3]. This implies that it is not straightforward to represent a (full) logical variable within the π-calculus setting because a logical variable has no directions in terms of read and write operations. In contrast, our approach restricts FGHC to its subset that can be embedded into PPC.

7 Concluding Remarks

This work shows how to embed moded Flat GHC into polyadic π-calculus (PPC), and $FGHC^-$ has been introduced as the source language for translation. Through the translation steps, the various features of moded Flat GHC, such as moded logical variables, passive and active unification, predicate invocation, concurrency and indeterminism, have been reexamined in the terminology of PPC. A translator, which generates a PIC program [9] from an $FGHC^-$ program, has been implemented in the KLIC system [1]. PIC is a programming language that was designed based on PPC. The specification of the two built-in predicates for shared data manipulation, *swap_shared_data* and *port*, has been demonstrated. By describing the built-in predicate specifications in PPC, we could clearly understand the resemblance and the difference between the two predicates.

The author thinks that the embedding into PPC is a basis for a new operational semantics for moded Flat GHC. For instance, this new operational semantics can be used for specifying new built-in predicates and provide a theoretical foundation for program transformation for efficient execution.

Acknowledgements: The author thanks Professor Kazunori Ueda for his valuable advice about moded Flat GHC. Also thanks to Dr. Benjamin Pierce for developing the PIC and the PICT systems, and Dr. Rikio K. Onai of NTT for continuously encouraging the author.

References

[1] T. Chikayama, T. Fujise, and D. Sekita. A portable and efficient implementation of KL1. In *Proc. 6th PLILP '94* (1994).

[2] S. Haridi, S. Janson, J. Montelius and M. Nilsson. Ports for Objects in Concurrent Logic Programs (DRAFT). *Unpublished document* (Dec. 1991).

[3] M. Henz, G. Smolka and J. Würtz. Oz – A Programming Language for Multi-Agent Systems, In *Proc. of IJCAI'93*, pp.404–409 (1993).

[4] K. Honda and V. Vasconcelos. Guarded Horn Clauses, *unpublished* (1990).

[5] B. Li. A π-calculus Specification of Prolog. In *Proc. of ESOP 1994* (1994).

[6] R. Milner. Functions as Processes. In *Proc. of 17th Int'l Colloquium on Automata, Languages and Programming*, LNCS 443 (1990).

[7] R. Milner. *The Polyadic π-Calculus: a Tutorial.* ECS-LFCS-91-180, University of Edinburgh (1991).

[8] S.-O. Nyström. Variable-Free Execution of Concurrent Logic Languages. In *Proc. of NACLP'89*, pp.536–552 (1989).

[9] B. Pierce. *Programming in the Pi-Calculus.* Available through anonymous FTP from ftp.dcs.ed.ac.uk (1993).

[10] E. Shapiro and S. Safra. Multiway Merge with Constant Delay in Concurrent Prolog. *New Generation Computing*, Vol. 4, pp.211–216 (1986).

[11] G. Smolka. *A Foundation for Higher-order Concurrent Constraint Programming.* RR-94-16, DFKI (1994).

[12] K. Ueda and T. Chikayama. Design of the Kernel Language for the Parallel Inference Machine. *The Computer Journal*, Vol.33, No.6, pp.494–500 (1990).

[13] K. Ueda and M. Morita. Moded Flat GHC and Its Message-Oriented Implementation Technique. to appear in *New Generation Computing*, OHMSHA, LTD. (1994).

[14] D. Walker. π-Calculus Semantics of Object-Oriented Programming Languages. In *Proc. of TACS'91*, LNCS 526 (1991).