The Buffered $\pi$-Calculus: A Model for Concurrent Languages

Xiaojie Deng$^1$, Yu Zhang$^2$, Yuxin Deng$^1$, and Farong Zhong$^3$

1 Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai, China {lvchaxj, yuxindeng}@gmail.com
2 Institute of Software, Chinese Academy of Sciences, Beijing, China yzhang@ios.ac.cn
3 Department of Computer Science, Zhejiang Normal University, Zhejiang, China zfr@zjnu.cn

Abstract. Message-passing based concurrent languages are widely used in developing large distributed and coordination systems. This paper presents the buffered $\pi$-calculus — a variant of the $\pi$-calculus where channel names are classified into buffered and unbuffered: communication along buffered channels is asynchronous, and remains synchronous along unbuffered channels. We show that the buffered $\pi$-calculus can be fully simulated in the polyadic $\pi$-calculus with respect to strong bisimulation. In contrast to the $\pi$-calculus which is hard to use in practice, the new language enables easy and clear modeling of practical concurrent languages. We encode two real-world concurrent languages in the buffered $\pi$-calculus: the (core) Go language and the (Core) Erlang. Both encodings are fully abstract with respect to weak bisimulations.

Keywords: process calculus, formal model, full abstraction

1 Introduction

Concurrent programming languages become popular in recent years thanks to the large demand of distributed computing and the pervasive exploitation of multi-processor architectures. Unlike the shared-memory concurrency model, which is now mainly used on multi-processor platforms, message passing based concurrent languages are particularly popular in developing large distributed, coordination systems. Indeed, quite a few real-world concurrent languages are intensively used in industry. The most well-known languages are probably Erlang, developed by Ericsson [1], and the much younger language Go, developed by Google [6]. Both languages achieve their asynchronous communication via order-preserving message passing.

On the other side, the $\pi$-calculus [11,15] has shown its success in modeling and verifying both specifications and implementations. Its asynchronous variant [18] is a good candidate as the target formal model. Despite the fact that it is called asynchronous, communication in the asynchronous $\pi$-calculus is however synchronous. It is shown in [2] that the communication modelled by the asynchronous $\pi$-calculus is equivalent to message passing via bags — senders put
messages into some bags, and receivers may get arbitrary messages from these bags. This result indicates that additional effort should be made to respect the order of the messages, which is adopted in the implementation of many concurrent languages.

In view of this, we may expect a formal model where asynchronous communication is supported natively. In fact, our primary goal is to achieve a formal model by which we can easily define a formal semantics of Go and do verification on top of it. The developers of Go claim that the concurrency feature of Go is rooted in CSP [7], while we show that the $\pi$-calculus should be an appropriate model for Go as CSP does not support transmission of channel names over channels.

In the spirit of the name passing mechanism of the $\pi$-calculus and the channel type of the Go language, we extend the $\pi$-calculus by introducing a special kind of names, each associated with a first-in-first-out buffer. We call these names buffered names. Communication along buffered names is asynchronous, while that along unbuffered (normal) names remains synchronous. We call this variant language the buffered $\pi$-calculus, and abbreviate it as the $\pi_b$-calculus.

We develop the $\pi_b$-calculus by defining its operational semantics as a labelled transition system and supplying an encoding into the polyadic $\pi$-calculus. We also present translations of the languages Go and Erlang into the $\pi_b$-calculus and show that the model is sufficient and relatively easier for modeling real-world concurrent languages.

1.1 Related Work

Beauxis et al introduced the $\pi_B$-calculus in order to study the asynchronous nature of the asynchronous $\pi$-calculus [2]. Their asynchronous communication is achieved via explicit use of buffers. In case that the buffers are ordered structures such as queues or stacks, the asynchronous communication modelled by $\pi_B$ differs from that by the asynchronous $\pi$-calculus. While communication in the $\pi_B$-calculus is always asynchronous, we keep both synchronous and asynchronous communication in the $\pi_b$-calculus, through different types of names.

Encoding programming languages in process calculus have been studied by many researchers. Milner defines the semantics of a non-trivial parallel programming language by a translation into CCS in [9]. In [17], a translation from a parallel object oriented language to the minimal $\pi$-calculus is presented. The correctness of the translation is justified by the operational correspondence between units and their encodings. Our treatments to the Go language follows the approach in [17]. In addition, we show a full abstraction theorem, namely equivalent Go programs are translated into equivalent $\pi_b$ processes.

For functional languages, Noll and Roy [12] presented an initial translation mapping from a Core Erlang [4] to the asynchronous $\pi$-calculus. Later on they [14] improved the translation by revising the non-deterministic encoding of pattern matching based expressions, and by adding the encoding for tuples. Their translations, however, are not sound in the sense that the order of messages is not always respected. By modelling the mailbox structure explicitly by
buffered names in the $\pi_b$-calculus, we obtain a more accurate encoding which is fully abstract with respect to weak bisimulation.

1.2 Outline

The rest of the paper is structured as follows. Section 2 presents the syntax and semantics of the $\pi_b$-calculus and a simple encoding in the polyadic $\pi$-calculus [10]. We show that this encoding preserves the strong bisimulation relation. In Section 3 we define a formal semantics for Go and present an encoding of Go in the $\pi_b$-calculus. Section 4 is devoted to Core Erlang, in which an improved encoding is presented. And finally, Section 5 concludes the paper.

2 The $\Pi_b$-Calculus

We assume an infinite set $\mathcal{N}$ of names, ranged over by $a, b, c, d, x, y$. Processes are defined by the following grammar.

$$P, Q, \ldots := \sum_{i \in I} \pi_i.P_i \mid P \mid Q \mid (\nu c : n)P \mid (\nu c)P \mid !P$$

where $\pi = c(x) \mid \tau(d) \mid \tau$.

Most of the syntax is standard: $\sum_{i \in I} \pi_i.P_i$ is the guarded choice ($I$ is finite), which behaves nondeterministically as one of its components $\pi_j.P_j$ for some $j \in I$; composition $P|Q$ acts as $P$ and $Q$ running in parallel; $!P$ is the replication of process $P$; Prefixes $c(x)$ and $\tau(d)$ are input and output along name $c$, and $\tau$ is the silent action. We write $0$ for the empty guarded choice, it is the process which can do nothing.

The $\pi_b$-calculus extends the $\pi$-calculus in the fact that names can be buffered or unbuffered. Unbuffered names are names in the $\pi$-calculus, and buffered names have the buffer attribute specified by a buffer store, denoted by $B$, is a partial function from buffered names to pairs $(n, l)$, where $n$ is a positive integer representing the capacity of the buffer, and $l$ is a list of names in the buffer, with the same order. Both $(\nu c)P$ and $(\nu c : n)P$ are called new processes. The (standard) new process $(\nu c)P$ specifies that $c$ (whether buffered or unbuffered) is a local name in $P$. The extended new process $(\nu c : n)P$ creates a local buffered name $c$, whose associated buffer has the capacity $n$ for asynchronous communication inside $P$. Notice that $(\nu c)P$ only says that the name $c$ is local and does not imply that $c$ is unbuffered — $c$ can be a buffered name whose buffer is already created in the buffer store.

Input process $c(x).P$ and output process $\tau(d).P$ can communicate with each other along name $c$ when they run in parallel. If $c$ is an unbuffered name, the communication is synchronous and happens as in the $\pi$-calculus: the object $d$ is passed from the output side to the input side. If $c$ is a buffered name, then the communication becomes asynchronous: the output process simply puts $d$ into the buffer of $c$ if it is not full and continues, or blocks if the buffer is full; the
input process retrieves the oldest value from the buffer of \( c \) if it is not empty and continues, or blocks if the buffer is empty.

As usual, we write \( \tilde{c} \) for a sequence of names, and abbreviate \((\nu c_1)\ldots(\nu c_n)\) \( P \) to \((\nu c_1\ldots c_n)\) \( P \). A name \( x \) is bound if it appears in input prefix, otherwise it is free. We write \( P\{\tilde{c}/\tilde{x}\} \) for the process resulting from simultaneously substituting \( c_i \) for each free \( x_i \) in \( P \). The newly created name \( c \) in \((\nu c : n)P\) or \((\nu c)P\) are local names. A name is global if it is not localized by any new operator. We use \( ln(P) \) and \( gn(P) \) for the set of local names and global names occurring in \( P \).

Throughout the development of the paper, we assume the following De Barendregt name convention:

Local names are different from each other and from global names.

For instance, we shall never consider processes like \( \pi(c). (\nu a) P \) or \( (\nu a)(\nu a) P \). We note that this convention is dispensable and we simply adopt it to make the presentation of the calculus simple and clean. One can also remove the convention and use syntactic rules to manage name conflicts, but dealing with names in buffers can be very subtle.

A process can send a local name into a buffer. The fact that a name stored in buffers is local must be tracked, because it may affect the name scope when another process retrieves this name from the buffer. The convention also works for buffer stores. We shall discuss more on this when defining the operational semantics. Inside a buffer store, a value of the form \((\nu c)\) indicates that the name \( c \) was sent into the buffer when it was local. Given a buffer store \( B \), we write \( gn(B(b)) \) for the set of global names that occur in \( b \)'s buffer, and \( gn(B) = \bigcup_{b \in \text{dom}(B)} gn(B(b)) \). Similarly \( ln(B(b)) \) and \( ln(B) \) for local names in \( B(b) \) and \( B \).

The buffer store \( B\{c/d\} \) is obtained by substituting \( c \) for each \( d \) in \( B \).

We say a process \( Q \) is guarded in \( P \), if every occurrence of \( Q \) in \( P \) is within some prefix process. Intuitively, a guarded process cannot affect the behavior of its host process until the action induced by its guarding prefix is performed. New operators are guarded in \( P \) if all new processes are guarded in \( P \).

The structural congruence \( \equiv_B \) with respect to the buffer store \( B \) is defined as the smallest congruence relation over processes satisfying the laws in Table 1.

| \( P \equiv_B Q, \) if | \( Q \) obtained from \( P \) by renaming bound names, or local names not occurring in \( B \). |
| \( P \mid Q \equiv_B Q \mid P; P \mid (Q \mid R) \equiv_B (P \mid Q) \mid R; P \mid 0 \equiv_B P. \) |
| \( !P \equiv_B P \mid !P. \) |
| \( (\nu c)(\nu d)P \equiv_B (\nu d)(\nu c)P. \) |
| \( (\nu c)0 \equiv_B 0, \) if \( c \notin ln(B). \) |
| \( (\nu c)(P \mid Q) \equiv_B (\nu c)P \mid Q, \) if \( c \notin ln(B) \land c \notin gn(Q) \) |

Table 1. Structural Congruence
most" level.

Buffer store \( B \) is valid for process \( P \) if each local name of \( B \) appears in some new operator occurring at the outermost level of \( P \), i.e., for every \( c \in \text{ln}(B) \), \( P \equiv_B (c)P' \) for some \( P' \).

### 2.1 Operational Semantics

| Rule | Description |
|-------|-------------|
| **IU** | \( c \not\in \text{dom}(B) \) |
| | \( c(x).P, B \xrightarrow{\text{in}(d)} P(d/x), B \) |
| **OU** | \( \tau \) |
| | \( P, B \xrightarrow{\text{output}} P, B \) |
| **OPEN** | \( \sigma \) |
| | \( P, B \xrightarrow{\sigma} P', B' \) |
| **IB** | \( b(x).P, B \xrightarrow{\text{in}(d)} P(d/x), B[b \mapsto (n,l)] \) |
| | \( B(b) = (n, [d] : l) \) |
| **OB** | \( \tau \) |
| | \( B(b) = (n,l); \ 1l < n; \ b \notin \text{ln}(P) \) |
| **OBG** | \( \tau \) |
| | \( B(b) = (n,[d]:l); \ b \notin \text{ln}(P) \) |
| **SUM** | \( j \in I; \ \pi_j.P, B \xrightarrow{\alpha} P', B' \) |
| | \( \sum_{i \in I} \pi_i.P, B \xrightarrow{\alpha} P', B' \) |
| **COM** | \( P, B \xrightarrow{\text{in}(d)} P', B'; \ Q, B \xrightarrow{\text{in}(d)} Q', B'; \ c \notin \text{dom}(B) \) |
| | \( P | Q, B \xrightarrow{\alpha} P' | Q', B' \) |
| **PAR** | \( P \equiv_B P' \) |
| | \( P \equiv_B P' \) |
| **NEW** | \( (\nu b:n).P, B \xrightarrow{\nu b} (\nu b)P, B[b \mapsto (n, [ ])] \) |
| | \( (\nu b:n).P, B \xrightarrow{\nu b} (\nu b)P, B[b \mapsto (n, [ ])] \) |
| **NEWB** | \( (\nu b:n).P, B \xrightarrow{\nu b} (\nu b)P, B[b \mapsto (n, [ ])] \) |

Table 2. Operational Semantics of \( \pi_b \)

The (early) transition semantics of \( \pi_b \) is given in terms of a labelled transition system generated by the rules in Table 2. The transition rules are of the form \( P, B \xrightarrow{\alpha} P', B' \), where \( P, P' \) are processes, \( B, B' \) are buffer stores and \( \alpha \) is an action, which can be one of the forms: the silent action \( \tau \), free input \( c(d) \), free output \( \text{in}(d) \) or bound output \( \text{out}(\nu d) \). We write \( n(\alpha) \) for the set of names occurring in \( \alpha \).

These rules are compatible with the transition rules for the \( \pi \)-calculus. IU and OU are rules for unbuffered names and synchronous communication is specified by COM. IB and OB define the asynchronous communication along buffered names: \( b(x).P \) performs a \( \tau \) action by receiving the “oldest” name \( d \) from \( b \)'s buffer, while \( \text{in}(d).P \) performs a \( \tau \) action by inserting \( d \) into \( b \)'s buffer. Communication along buffered names is asynchronous because it involves two transitions (IB and OB) and other actions may occur between them.

IBG and OBG indicate that a buffer store itself may have actions. If \( b \) is a global buffered name, that is \( (\nu b) \) does not occur in \( P \), then we can insert names
X. Deng, Y. Zhang, Y. Deng, and F. Zhong

to or receive names from b’s buffer directly. In **NEW** and **OPEN**, the substitutions on the buffer store are for the sake of validity. **NEWB** is the rule for the extended new process. After creating an empty buffer for b, the capacity parameter n is dropped, leaving the new operator indicating that b is a local name.

The **PAR** rule describes how processes can progress asynchronously, which typically happens with buffered names. However, unlike in the π-calculus, where we have open/close rules to manage name scope extension, in the πb-calculus, it is hard (perhaps impossible) to define an appropriate close rule because when a local name is exported to a buffer, it becomes hard to track which process will retrieve the name so as to determine the name scope. For instance, consider the process $P_1 | P_2 | P_3$ where $P_1 = (\nu a)(b(a)).P_1'$, $P_2 = b(y) \cdots$, $P_3 = b(z) \cdots$ and a valid buffer store $B = [b \mapsto (2, [ ])]$. In the πb-calculus, $P_1$ inserts the local a into b’s buffer by a τ action, then it can possibly be received by $P_2$ or $P_3$, hence tracking the scope of a becomes very hard. Our solution here is to prevent processes from inserting local names into buffers when they are running in parallel with other processes. For processes like the above example, we extend the scope of a to the entire process by structural congruence laws and obtain a process in the form $(\nu a)(b(a)).P_1'|P_2|P_3$ thanks to the name convention. This avoids the scope problem.

We have adopted the name convention which simplifies the definition of the labeled transition system. Dealing with names with buffers is subtle and the transition rules without the name convention are discussed in the next subsection.

The following proposition says that transition rules preserve buffer validity:

**Proposition 1.** If $B$ is valid for process $P$ and we have the transition $P, B \xrightarrow{\alpha} P', B'$, then $B'$ is valid for $P'$.

As in the π-calculus, strong bisimulation over the set of πb processes can be defined as follows.

**Definition 2.** A symmetric binary relation $R$ over πb processes is a bisimulation, if whenever $(P, B_P)R(Q, B_Q)$ and $(P, B_P) \xrightarrow{\alpha} (P', B'_P)$,

$$\exists (Q', B'_Q) \cdot (Q, B_Q) \xrightarrow{\alpha} (Q', B'_Q) \land (P', B'_P)R(Q', B'_Q)$$

Strong bisimilarity $\sim$ is the largest strong bisimulation over the set of πb processes. $(P, B_P)$ and $(Q, B_Q)$ are strongly bisimilar, written as $(P, B_P) \sim (Q, B_Q)$, if they are related by some strong bisimulation.

**Transition Rules without Name Conversion** As mentioned above, some transition rules require extra conditions to deal with name conflict without the name conversion. These rules are shown in Table 3.

The problem is how can we determine a local name in the buffers refers to which local name of the process. For instance, suppose $P = (\nu c)(\nu c)P'$ with valid buffer store $B = [b \mapsto (5, [\nu c])]$. We have no idea the $\nu c$ in $B$ refers to which one of the two local cs in $P$. Therefore we first assume the local names in buffers
Open*  \[ P, B[c/νc] \xrightarrow{τ} P', B' \; ; \; c \notin \text{n}(α); \; c \notin \text{gn}(B); \; c \notin \text{ln}(B') \]

\( b \notin \text{dom}(B) \)

NewB*  \[ (vb : n) P, B \xrightarrow{(vb)} P[b \xrightarrow{(n,[])}] \]

New*  \[ P, B[c/νc] \xrightarrow{π(νc)} P', B' \; ; \; c \notin \text{gn}(B) \]

\( (νc)P, B \xrightarrow{π(νc)} P', B' \)

Table 3. Operational Semantics without Name Conversion

are localized by the “outermost” and “leftmost” new operator of the process, and add additional conditions to transition rules to respect the assumption.

In New*, for those global \( c \) in \( B \), they are semantically different from the local \( c \) in \( P \). For example \( P = b(c), (νc)b(c) \), with \( B = [b \xrightarrow{(5,[])}] \). A \( τ \) transition inserts the global \( c \) into \( b \)'s buffer, and we have \( P' = (νc)b(c) \), with \( B' = [b \xrightarrow{(5,[c])}] \). At this point, we intend to insert the local \( c \) into \( b \)'s buffer, this local \( c \) is apparently different from the global \( c \) already in the buffer. We add the condition \( c \notin \text{gn}(B) \) to enforce a renaming of the local \( c \) of \( P \) before the insertion. The same discussion applies to the same extra condition of Open*.

In rare cases, another condition is required for New*. Suppose, for instance, \( P = (νc)(νc)b(c), b(c) \), with \( B = [b \xrightarrow{(5,[])}] \). According to existing New rule, the process may perform a \( τ \) action inserting a local \( c \) into \( b \)'s buffer and become \((νc)(νc)b(c)\), with buffer store changed to \([b \xrightarrow{(5, [νc])}]\) where the local \( c \) is actually localized by the second \((νc)\). But by our assumption, it would refer to the first local \( c \) of the process. We avoid this inconsistency by introducing the condition \( c \notin \text{ln}(B') \).

The condition \( b \notin \text{dom}(B) \) in NewB* guarantees that a fresh buffered name is used.

Examples We demonstrate these transition rules by showing some examples. The following example illustrates the asynchronous communication by buffered names.

Example 3. \( P = (νa)\overline{b}(a), a(x) \mid b(y), \overline{τ}(c) \mid R \) and \( B = [b \xrightarrow{(5,[])}] \)

\( (νa)\overline{b}(a), a(x) \mid b(y), \overline{τ}(c) \mid R, \quad B[b \xrightarrow{(5,[])}] \)

\( \equiv_B (νa)(\overline{b}(a), a(x) \mid b(y), \overline{τ}(c) \mid R), \quad B[b \xrightarrow{(5,[])}] \)

New*  \( (νa)(a(x) \mid b(y), \overline{τ}(c) \mid R), \quad B[b \xrightarrow{(5,[])}] \)

New  \( (νa)(a(x) \mid \overline{τ}(c) \mid R), \quad B[b \xrightarrow{(5,[])}] \)

\( P \) is a parallel composition, we may not use PAR immediately as it contains an unguarded new process. After moving the new operator to the outermost level, we may apply the New rule which induces a \( τ \) transition ‘sending’ the local \( a \) into \( b \)'s buffer. Notice that the local \( a \) is not directly inserted into the buffer, but in
a substituting way — $a$ is inserted into $b$’s buffer in the premise of $\text{NEW}$, then $a$ is replaced by $\nu a$ in the conclusion. The second $\tau$ step describes the ‘receiving’ of a local name from a buffer.

The next three examples illustrate the extra $\text{NEW}^*$ rule.

**Example 4.** $P = \overline{b}(a). (\nu a)\overline{b}(a) \overline{b}(a)$ and $B = [b \mapsto (5, [ ])]$

$$
\begin{align*}
\text{IB} & \xrightarrow{\tau} (\nu a)\overline{b}(a), B[b \mapsto (5, [ ])] = B' \\
\equiv & \text{IB}^* (\nu a'')\overline{b}(a'), B[b \mapsto (5, [ ])]
\end{align*}
$$

The first $\tau$ step follows from IB. At this point, the local name $a$ occurs free in buffer store, hence an $\alpha$-conversion is required. After renaming $a$ to $a'$, a second $\tau$ transition ‘sending’ the local name to $b$’s buffer.

**Example 5.** $P = (\nu a)\overline{b}(a). (\nu a)\overline{b}(a)$ and $B = [b \mapsto (5, [ ])]$

$$
\begin{align*}
\text{NEW}^* & \xrightarrow{\tau} (\nu a)\overline{b}(a). (\nu a)\overline{b}(a), B[b \mapsto (5, [ ])] = B' \\
\equiv & \text{NEW}^* (\nu a)(\nu a')\overline{b}(a'), B[b \mapsto (5, [ ])]
\end{align*}
$$

After inserting the local $a$ into the buffer, the process contains two outermost new operators and they are syntactically the same (semantically different). Following $\text{NEW}^*$, we first determine the action of $(\nu a)\overline{b}(a)$ with $B[b \mapsto (5, [a])]$. Since the local name $a$ occurs free in the buffer store, an $\alpha$-conversion is required. After that, a $\tau$ transition results in $(\nu a')0$ with buffer store $B[b \mapsto (5, [a, \nu a'])]$. Finally, all the $a$ in buffer store are modified back to $\nu a$.

**Example 6.** $P = (\nu a)(\nu a)\overline{b}(a) \overline{b}(a)$ and $B = [b \mapsto (5, [ ])]$

$$
\begin{align*}
(\nu a)\overline{b}(a), B[b \mapsto (5, [ ])] & \xrightarrow{\tau} (\nu a)\overline{b}(a), B[b \mapsto (5, [ ])] = B' \\
(\nu a)(\nu a)\overline{b}(a), B[b \mapsto (5, [ ])] & \equiv \text{NEW}^* (\nu a)(\nu a')\overline{b}(a'), B[b \mapsto (5, [ ])]
\end{align*}
$$

$P$ contains two outermost new operators at the beginning. We first determine the action of $(\nu a)\overline{b}(a)$ with buffer $B[b \mapsto (5, [ ])]$. A $\tau$ transition leads to $(\nu a)\overline{b}(a)$ with $B[b \mapsto (5, [\nu a])]$ where the local $a$ actually refers to the second $\nu a$ of the original process. Hence an $\alpha$-conversion of the second $\nu a$ is required to distinguish itself with the first one. This requirement is captured by $a \not\in \text{ln}(B')$ in $\text{NEW}^*$. 
An interlude of Close Rule and Structure Congruence In $\pi$ calculus, including which law in structure congruence and which other rule in transition rules is a trade-off. This phenomenon also exists in our $\pi_b$ calculus. One may ask for including a similar $\text{Close}_\pi$ rule, which generates a $\tau$ action by synchronizing an input and a bound output action, and omitting those scope extension laws in structure congruence.

$$\text{Close}_\pi \quad \frac{P \xrightarrow{\tau[(\nu a)]} P';\ Q \xrightarrow{b(a)} Q'}{P \parallel Q \xrightarrow{\nu a} (\nu a)(P' \parallel Q')}$$

In $\text{Close}_\pi$, the local name moves to the action in the premise and to the outer-most level in the conclusion. What should the rule be in $\pi_b$?

Suppose $P$ performs a $\tau$ action sending a local name $a$ to $b$’s buffer and becomes $P'$, and at some point in the future $Q$ performs a $\tau$ action receiving this local name from the buffer and becomes $Q'$, the scope of $a$ should contain both $P'$ and $Q'$. But if $P''$ derives to other process during the period between the two silent actions, how can we determine which processes we should encompass by the new operator $\nu a$.

In the $\pi$-calculus, communications are synchronous, input process would proceed unless a complement (bound) output process is ready, and vice versa. However, in the $\pi_b$-calculus, communications along buffered names are asynchronous, other actions may occur between the two transitions. For this reason, we choose scope extension laws instead of some $\text{Close}_\pi$ rules.

2.2 Encoding in the Polyadic $\Pi$-Calculus

We demonstrate an encoding of the $\pi_b$-calculus in the polyadic $\pi$-calculus.

Intuitively, a $\pi_b$ name $c$ is encoded into a pair of $\pi$ names $(c_1, c_2)$ by the injective name translation function $N$. In the name pair, $c_1$ is called the input name and $c_2$ the output name of $c$. In addition, input and output names for unbuffered names are identical, but not for buffered names. The two translation names of buffered name $b$ are exactly the names along which a buffer process modelling the buffer of $b$ receives and sends values.

1. If $a$ is a unbuffered name, then $a_1 = a_2$ where $N(a) = (a_1, a_2)$.
2. If $b$ is a buffered name, then $b_1 \neq b_2$ where $N(b) = (b_1, b_2)$.
3. For any two names $c \neq d$, $c_1 \neq d_1 \land c_1 \neq d_2 \land c_2 \neq d_1 \land c_2 \neq d_2$.

Table 4. Name Translation Function $N$

|  |  |
|---|---|
| 1. If $a$ is a unbuffered name, then $a_1 = a_2$ where $N(a) = (a_1, a_2)$. |  |
| 2. If $b$ is a buffered name, then $b_1 \neq b_2$ where $N(b) = (b_1, b_2)$. |  |
| 3. For any two names $c \neq d$, $c_1 \neq d_1 \land c_1 \neq d_2 \land c_2 \neq d_1 \land c_2 \neq d_2$. |  |

The buffer process is defined in Table 5. Intuitively speaking, $F_{n,L}(b_1, b_2)$ is the $\pi$ representation of $b$’s buffer, where $n$ denotes the capacity and $L$ is a list of $\pi$ name pairs. This process may further receive a pair of names along its second
$F_{n,L}(b_1, b_2) \overset{def}{=} b_2(x_1, x_2).F_{n,L_1}(b_1, b_2)$ \quad (|L| = 0)

where $L_1 = [(x_1, x_2)]$

$F_{n,L}(b_1, b_2) \overset{def}{=} b_2(x_1, x_2).F_{n,L_1}(b_1, b_2) + \overline{b_1}(v_1, v_2).F_{n,L_2}(b_1, b_2)$ \quad (0 < |L| < n)

where $L = L_1 :: [(x_1, x_2)]; L_2 = [(v_1, v_2)] :: L_2$

$F_{n,L}(b_1, b_2) \overset{def}{=} b_2(v_1, v_2).F_{n,L_2}(b_1, b_2)$ \quad (|L| = n)

where $L = [(v_1, v_2)] :: L_2$

Table 5. Buffer Process $F_{n,L}$

parameter $b_2$ if $L$ is not full ($|L| < n$) or send a pair of names along its first parameter $b_1$ providing $L$ is not empty.

The translation function $[ [ \cdot ] ]$ takes a $\pi_b$ process and a valid buffer store as parameter and returns a single $\pi$ process. The encoding of a buffer store is a composition of buffer processes each representing a buffered name’s buffer. For processes, the encoding differs from the original process in the new operators and prefixes. A new operator is encoded into two new operators localizing the pair of translation names. The encoding of input prefix $c(x)$ is also an input prefix but the subject is $c$’s input name $c_1$, while the encoding of output prefix $\overline{c}(d)$ has the output name $c_2$ as the subject. Finally, in the encoding of an extended new process $(\nu b : n)P'$, a buffer process representing $b$’s buffer is added.

The action translation function $M$ maps $\pi_b$ actions to corresponding $\pi$ actions, it is defined similar to the encoding of prefixes.

With an abuse of notation, we also write $[ [P] ]$ and $[ [B] ]$ for the encoding of $P$ and $B$ respectively. The translation function $[ [\cdot] ]$, along with the bijective action translation function $M$ are defined in Table 6.

The following properties are apparent. Substitutions can be postponed until after the translation.

**Proposition 7.** For a process $P$, and the translation function $[ [\cdot] ]$

$[ [P(c/x)] ] = [ [P] ]_{c_1, c_2/x_1, x_2}$

where $N(x) = (x_1, x_2)$ and $N(c) = (c_1, c_2)$

And structure congruent processes have the ‘same’ encodings.

**Proposition 8.** If $P \equiv_B Q$, then $[ [P, B] ] \equiv [ [Q, B] ]$

These propositions can by proved by induction on the structure of $P$

**Full Abstraction** The following two lemmas show that transitions of a $\pi_b$ process can be simulated by its encoding, and no more transition is introduced by the encoding.

**Lemma 9.** Suppose $(P, B) \overset{\alpha}{\rightarrow} (P', B')$, then $[ [P, B] ] \overset{M(\alpha)}{\rightarrow} [ [P', B'] ]$. 
Lemma 10. Suppose $[P, B] \xrightarrow{M(\alpha)} R$, then $(P, B) \xrightarrow{\alpha} (P', B')$ and $R = [P', B']$.

Strong bisimulation relation is retained in the translation.

Lemma 11. If $(P, B_P) \sim (Q, B_Q)$, then $[P, B_P] \sim [Q, B_Q]$.

Proof. We show the following relation $\mathcal{R}$ is a strong bisimulation.

$$\mathcal{R} = \left\{ ([P, B_P], [Q, B_Q]) \mid (P, B_P) \sim (Q, B_Q) \right\}$$

Suppose $[P, B_P] \xrightarrow{M(\alpha)} R$, then by lemma 10

$$R = [P', B'_P']$$

and also $([P', B'_P'], [Q', B'_Q']) \in \mathcal{R}$ because $(P', B'_P') \sim (Q', B'_Q')$.

The other direction is the same.

Lemma 12. If $[P, B_P] \sim [Q, B_Q]$, then $(P, B_P) \sim (Q, B_Q)$.
Proof. We show the following relation $\mathcal{R}$ is a strong bisimulation.

$$\mathcal{R} = \left\{ ((P, B_P), (Q, B_Q)) \mid [P, B_P] \sim [Q, B_Q] \right\}$$

Suppose $(P, B_P) \xrightarrow{\alpha} (P', B'_P)$, then by lemma 9

$$[P, B_P] \xrightarrow{M(\alpha)} [P', B'_P]$$

Since $[Q, B_Q] \xrightarrow{M(\alpha)} R$, by lemma 10

$$R = [Q', B'_Q]$$

and also $((P', B'_P), (Q', B'_Q)) \in \mathcal{R}$ because $[P', B'_P] \sim [Q', B'_Q]$.

The other direction is the same. $\Box$

It follows that the encoding preserves strong bisimulation.

Theorem 13. $(P, B_P) \sim (Q, B_Q)$ if and only if $[P, B_P] \sim [Q, B_Q]$.

3 The Go Programming language

The Go programming language is a general purpose language developed by Google to support easy and rapid development of large distributed systems. It is a relatively young language that inherits many good qualities of its ancestor while at the same time introduces dozens of innovations for efficient and effective programming. One of the most fascinating innovations is the concurrency feature which extremely simplifies the construction of concurrent applications. This section presents a formal operational semantics of the (core) Go language and a fully abstract encoding in the $\pi_b$-calculus.

The syntax of a core of Go is presented in Table 7. An online specification of Go can be found at its website [6].

The channel type, coupled with the concept called Go-routine, constitutes the core of Go’s concurrency system. Channel types are of the form `chan t`, where `t` is called the element type. Channels (ch) are first-class values of this language, and they are created by the make expression `make(chan t, n)`, where `chan t` specifies the channel type and the integer `n` specifies the size of the channel buffer. Notice that `n` must be non-negative and if it is zero, the created channel will be a synchronous channel.

Go-routines are similar to OS threads but much cheaper. A Go-routine is launched by the statement `go f(v1...vn)`. The function body of `f` will be executed in parallel with the program that executes the `go` statement. When the function completes, this Go-routine terminates and its return value is discarded.

Communication among Go-routines is achieved by sending and receiving operations on channels. Sending statement `ch<-v` sends `v` to channel `ch`, while

---

1 Goole has claimed that Go is used in production now. Specifically, the website of Go ([golang.org](http://golang.org)) and the download site ([dl.google.com](http://dl.google.com)) are written in Go.
Types: \[ t ::= \text{int} \mid \text{chan } t \]

Expressions:
\[
e, e_1, e_2, \ldots ::= x \quad \text{Variable} \\
\quad | \quad n \quad \text{Integer} \\
\quad | \quad ch \quad \text{Channel} \\
\quad | \quad \text{make}(\text{chan } t, n) \quad \text{Channel creation} \\
\quad | \quad \leftarrow e \quad \text{Receiving}
\]

Statements:
\[
s, s_1, s_2, \ldots ::= x = e \quad \text{Assignment} \\
\quad | \quad e_1\leftarrow e_2 \quad \text{Sending} \\
\quad | \quad s_1; s_2 \quad \text{Sequential} \\
\quad | \quad \text{go } f(e_1 \ldots e_n) \quad \text{Go-routine} \\
\quad | \quad \text{select} \{ c_1 \ldots c_n \} \quad \text{Selection}
\]

where \( c_1, c_2, \ldots ::= \text{case } x = \leftarrow e : s \mid \text{case } e_1\leftarrow e_2 : s \)

Table 7. Syntax of the (core) Go

receiving \( \leftarrow ch \), regarded as an expression in Go, receives a value from \( ch \). Communication via unbuffered channels are synchronous. Buffered (non-zero sized) channels enable asynchronous communication. Sending a value to a buffered channel can proceed as long as its buffer is not full and receiving from a buffered channel can proceed as long as its buffer is not empty.

\textbf{select} statements introduce non-deterministic choice, but their clauses refer to only communication operations. A \textbf{select} statement randomly selects a clause whose communication is “ready” (able to proceed), completes the selected communication, then proceeds with the corresponding clause statement.

Without loss of generality, we stipulate that a Go program is a set of function declarations, each of the form

\[
\text{func } f(x_1 \ldots x_n) \{ s \}
\]

A Go program must specify a main function, which we shall refer to as \( f_{\text{start}} \) in the sequel, as the entry point — running a Go program is equivalent to executing \( \text{go } f_{\text{start}}(\ldots) \) with appropriate arguments. For the sake of simplicity, we only consider function calls in \text{go} statements and we assume that all functions do not return values and their bodies contain no local variables other than function arguments.

### 3.1 Operational Semantics

The structural operational semantics of Go is defined by a two-level labelled transition system: the local transition system specifies the execution of a single Go-routine in isolation, and the global transition system describes the behavior of a running Go program.

We first define the evaluation of expressions. An expression configuration is a triple \( \langle e, \sigma, \delta_e \rangle \), where \( e \) is the expression to be evaluated, \( \sigma \) is the local store
mapping local variables to values, and $\delta_c$ is the channel store mapping channels to triples $(n,l,g)$, where $n$ is the capacity of the channel’s buffer, $l$ is a list of values in the channel buffer, and $g$ is a tag indicating whether the channel is local (0) or global (1). The transition rules between expression configurations $\Rightarrow_g$ are defined in Table 8, where actions can be either silent action $\tau$, or $r(ch,v)$ denoting receive action. We often omit $\tau$ from silent transitions.

| Rule | Description |
|------|-------------|
| **VAR** | $(x,\sigma,\delta_c) \Rightarrow_g (\sigma(x),\sigma,\delta_c)$ |
| **MAK** | $(\text{make(chan } \tau,n),\sigma,\delta_c) \Rightarrow_g (\text{ch},\sigma,\delta_c[\text{ch} \mapsto (n,\lambda,0)])$ |
| **RVE** | $(e,\sigma,\delta_c) \Rightarrow_g (e',\sigma,\delta_c)$ |
| **RvE** | $(\llcorner e,\sigma,\delta_c) \Rightarrow_g (\llcorner e',\sigma,\delta_c)$ |
| **RvB** | $\delta_c(ch) = (0,\lambda,g)$ |
| **RvU** | $\delta_c(ch) = (n,\llcorner v:g)$ |
| **RvU** | $\delta_c(ch) = (n,\llcorner v:0)$ |

Table 8. Transition Rules for Expressions

**VAR** retrieves the value of $x$ from local store $\sigma$. **MAK** creates a fresh local channel $ch$. Other rules concern receiving from channels. Once the channel expression is fully evaluated, the real receive begins following rules **RvU** and **RvB**. The value received from an unbuffered channel is indicated in the label, while the value received from a buffered channel is the “oldest” value of the channel’s buffer.

The local transition system defines transition rules between local configurations. A local configuration is a tuple $\langle s,\sigma,\delta_c \rangle$, where $s$ is the statement to be executed, $\sigma$ is the local store and $\delta_c$ is the channel store. Each Go-routine has its own local store, but the channel store is shared by all Go-routines of a running program. The local transition relation $\Rightarrow_g$ is presented in Table 9. Two additional actions can occur in local transition rules: $\alpha(ch,v)$ for message sending over channels and $\beta(f,v_1,\ldots,v_n)$ for Go-routine creation.

Subexpression evaluation in Go is strict and leftmost, and this evaluation strategy is specified by $\alpha$, $\beta_1$, $\beta_2$, $\gamma$ and $\delta$. For select statement, its subexpressions are those in its communication operations — the $e_1,e_2$ in case $e_1 \llcorner e_2 : s$ and the $e$ in case $x = \llcorner e : s$.

Rules **SdU** and **SdB** capture the behavior of sending over unbuffered and buffered channels respectively. Sending a value $v$ over an unbuffered channel $ch$ carries a sending label $\alpha(ch,v)$, while sending over buffered channels is silent and can proceed as long as the target channel buffer is not full. The **Go** rule says that a go statement does nothing locally and can always proceed with a transition with the $\beta$ label — the label is here simply for notifying the global configuration to generate corresponding Go-routines. **Ass** assigns $v$ to variable
The Buffered $\pi$-Calculus 15

Table 9. Local Transition Rules of Go
x. Seq1 and Seq2 specify the sequential execution. In SLR and SLS, the select statement picks the i-th clause.

Global transitions happen between global configurations which contain information of all running Go-routines. A global configuration, denoted by $A, A_1, \ldots$, is defined as a tuple $\langle \Gamma, \delta_c \rangle$, where $\Gamma$ is a multi-set of statement/local store pairs $(s, \sigma)$, of all running Go-routines, and $\delta_c$ is the channel store.

A global transition takes the form

$$\delta_f \vdash \langle \Gamma_1, \delta_{c_1} \rangle \xrightarrow{a} \langle \Gamma_2, \delta_{c_2} \rangle$$

where $\delta_f$ is a mapping from function names to function definitions. A Go program will start from an initial configuration $\langle \{s_{\text{start}}, \sigma_{\text{start}}\}, \delta_{\text{init}} \rangle$, where $s_{\text{start}}$ is the body of the main function $\text{start}$, $\sigma_{\text{start}}$ is the local store of $\text{start}$, and $\delta_{\text{init}}$ is the initial channel store. The global transition rules are listed in Table 10. A global action can be either $\tau$, $r(ch, v)$ or $a(ch, v)$.

| Rule | Description |
|------|-------------|
| LOC  | $\langle s, \sigma, \delta_c \rangle \xrightarrow{\tau} \langle s', \sigma', \delta_c' \rangle$ |
| COM  | $\delta_f \vdash \langle \Gamma \cup \{(s, \sigma)\}, \delta_c \rangle \xrightarrow{\text{seq} \{ch,v\}} \langle \Gamma \cup \{(s', \sigma')\}, \delta_c' \rangle$ |
| LGO  | $\langle s, \sigma, \delta_c \rangle \xrightarrow{\text{seq} \{f(x_1 \ldots x_m) \{y_f\}\} \text{\underline{GRB}}} \langle \Gamma \cup \{(s', \sigma)\}, \delta_c' \rangle$ |
| GRU  | $\delta_f \vdash \langle \Gamma \cup \{(s, \sigma)\}, \delta_c \rangle \xrightarrow{\text{seq} \{ch,v\}} \langle \Gamma \cup \{(s', \sigma)\}, \delta_c' \rangle$ |
| GRB  | $\delta_f \vdash \langle \Gamma, \delta_c \rangle \xrightarrow{\text{seq} \{ch,v\} \text{\underline{GSU}} \text{\underline{GSB}}} \langle \Gamma, \delta_c'[\{ch \mapsto \{v, l\} \}] \rangle$ |

Table 10. Global Transition Rules of Go

Loc specifies the independent transition of a single Go-routine. Asynchronous communication will also take this transition since RvB and SdB are both silent.
transitions. \texttt{LGo} creates a new Go-routine. \texttt{Com} defines the synchronous communication between two Go-routines over unbuffered channels. The rules \texttt{Loc}, \texttt{LGo} and \texttt{Com} all specify internal actions of a running program.

A Go program can communicate with the environment via global channels. \texttt{GRU}, \texttt{GSU1} and \texttt{GSU2} describe how a Go program interact with the environment via unbuffered channels, and \texttt{GRB}, \texttt{GSB1} and \texttt{GSB2} describe interactions via buffered channels. Because communication over buffered channels are asynchronous, the labels in \texttt{GRB}, \texttt{GSB1} and \texttt{GSB2} indicate how a global channel interacts with the environment. For instance, in \texttt{GRB} the label $\tau(ch, v)$ means that the channel (buffer) $ch$ receives a value $v$ from the environment. The two rules \texttt{GSU2} and \texttt{GSB2} also describe how a local channel is exposed to the environment and becomes a global channel, by communication upon global channels. The $\nu$ in the label is required only when the value is a local channel ($g' = 0$).

Let $t = \alpha_1 \ldots \alpha_n$ where each $\alpha_i$ is a global action, we write $\hat{t}$ for the action sequence obtained by eliminating all the occurrences of $\tau$ in $t$. We write $P, B \xrightarrow{\tau_g} P', B'$ if $P, B \xrightarrow{\alpha_1 \ldots \alpha_n \tau_g} P', B'$, and $P, B \xrightarrow{\tau_g} P', B'$ if $P, B \Rightarrow_g \alpha_1 \ldots \alpha_n \Rightarrow_g \ldots \Rightarrow_g \tau_g \Rightarrow_g P', B'$, where $\Rightarrow_g$ is the reflexive and transitive closure of $\xrightarrow{\tau_g}$.

**Definition 14.** A symmetric binary relation $\mathcal{R}$ over global configurations is a (weak) bisimulation if

$$A_1 \mathcal{R} A_2 \land A_1 \xrightarrow{\tau_g} A_1' \text{ then } \exists A_2'. \ A_2' \xrightarrow{\hat{\tau}_g} A_2' \land A_1' \mathcal{R} A_2'$$

Two global configurations are bisimilar, written as $A_1 \approx_g A_2$, if they are related by some bisimulation.

Two Go programs $gp_1, gp_2$ are bisimilar, if their initial global configurations (with the same $\delta_c$) are bisimilar.

### 3.2 Encoding

The encoding of Go in the $\pi_b$-calculus is achieved by the translation function $\llbracket \cdot \rrbracket_g(r)$, which maps Go expressions and statements to $\pi_b$ processes. The parameter $r$ is the name along which the result of an expression is returned or the termination of a statement is signaled. The translation function $\llbracket \cdot \rrbracket_g$ is defined in Table 11.

In the encoding, we use synchronous communication via local names to arrange the evolution order of $\pi_b$ processes. For instance, in \texttt{Recv}, the right hand side of the composition will not proceed unless the left hand side outputs along local name $r'$.

Process $\texttt{Var}(x, v)$ denotes variable $x$ whose current value is $v$. After inputting a pair of local names $(g, p)$, one can retrieve the associated value by communicating on $g$ or update the variable by communicating on $p$. Process $\texttt{LR}$ evaluates these non-fully evaluated expressions in an expression sequence in left-to-right order by synchronous communication on local names.
\[
V_{Dec} Var(x, v) = (\nu t)(\mathcal{T}(v) || l(t) . x(g, p) . (p(y). \mathcal{T}(z). \mathcal{T}(z)))
\]

\[
L_{RE} LR(v_1 . . . v_n, r) = \tau(v_1 . . . v_n)
\]

\[
LR(. . . e_1, e_2 . . . e_m, . . . r) = (\nu r_1 t_2 r_2 . . . t_m r_m)(
\[
[e_1]_y(r_1) \mid [e_2]_y(r_2) . . . [e_m]_y(r_m)
\]

| \tau_1(r_1) \mid \tau_2(r_2) . . . \tau_m(r_m). \tau(v_1 . . . v_n)
\]

\[
V_{A\ell} [n]_p(r) = \tau(n)
\]

\[
\begin{align*}
\text{VARI} [x]_p(r) & = (\nu g p)(\tau(g, p). g(z). \tau(z)) \\
\text{RECV} \langle e \rangle_s_p(r) & = (\nu r')([e]_p(r'))|r'(y, z). \tau(z)) \\
\text{MAKE} \langle \text{chan } \tau, 0 \rangle_p(r) & = (\nu a)\tau(a) \\
\text{MAKE} \langle \text{chan } \tau, n \rangle_p(r) & = (\nu b : n)\tau(b)
\end{align*}
\]

\[
V_{A\ell} [\text{nil}]_p(r) = \tau
\]

\[
\begin{align*}
\text{ASSI} [x = e]_p(r) & = (\nu r')(\langle e \rangle_p(r')|r'(z). (\nu g p)(\tau(g, p). \tau(z). \tau)) \\
\text{SEND} \langle e_1 \leftarrow e_2 \rangle_p(r) & = (\nu r')(\tau(\langle e_1, e_2, r' \rangle)|r'(y, z). \tau(z). \tau) \\
\text{SEQU} [s_1; s_2]_p(r) & = (\nu r')(\langle [s_1]_p(r')|r'. [s_2]_p(r)) \\
\text{Go} \langle g o f(e_1 \ldots e_n) \rangle_p(r) & = (\nu r')\tau(\langle e_1 \ldots e_n, r' \rangle)
\end{align*}
\]

\[
\begin{align*}
\text{SELECT} \{e_1 \ldots e_n\}_p(r) & = (\nu r')(\tau(\langle e_1 \ldots e_n, r' \rangle)
\end{align*}
\]

\[
\begin{align*}
| r'(y_1 \ldots y_n). \tau(y_1 \ldots y_n). \tau
\end{align*}
\]

\[
\begin{align*}
\text{SC} = \left\{
\begin{array}{ll}
y_1(v), [x = v; a]_p(r) & \text{c \equiv \text{case } x = \leftarrow e_i \!: s} \\
\tau(y_j), [x]_p(r) & \text{c \equiv \text{case } e_i \leftarrow e_j \!: s}
\end{array}
\right.
\end{align*}
\]

Table 11. Encoding of Go
MAKE returns the local name denoting the newly created channel. A receive operation corresponds to an input prefix in `recv`, while a send operation corresponds to an output prefix in `send`. For the `go` statement, after evaluating the argument expressions, these arguments are sent to the function to which `f` refers. The statement does not wait for the function, rather it outputs the termination signal along `r` immediately.

For `select`, suppose $e_1 \ldots e_k$ is these (fully and non-fully evaluated) subexpressions appearing in the communication operation of its clauses listed in lexical top-to-bottom and left-to-right order. The encoding first evaluates this expression sequence, followed by a guarded choice each of its constituent denotes a select clause. The use of guarded choice here seems unavoidable.

In the encoding, some prefixes and extended new operators are underlined. They are the most significant part and will be discussed later.

The translation function can be extended to a mapping from global configurations (with $\delta_f$) to $\pi_3$ processes. We write $[[A]]_g$ for the pair $(P, B)$, where $P$ is the encoding of $A$ and $\delta_f$, and $B$ is a valid buffer store inferred from channel store $\delta_c$. The extended translation function is shown in Table 12.

\[
\begin{align*}
\text{FDec} &\left[\text{func } f(x_1 \ldots x_n) \{s\} \right]_g = !f(z_1 \ldots z_n).(\nu x_1 \ldots x_n)(\text{Var}(x_1, z_1)| \ldots |\text{Var}(x_n, z_n))|(\nu r')(\delta)_g(r') \\
\text{LStr} &\left[s\left[\sigma\right]_g = \prod_{i=1}^n \text{Var}(x_i, \sigma(x_i)) \quad x_i \in \text{dom}(\sigma); \ n = |\text{dom}(\sigma)|
\right.
\text{FEnv} &\left[\delta_f\right]_g = \prod_{i=1}^n [\delta_f(f_i)]_g \quad f_i \in \text{dom}(\delta_f); \ n = |\text{dom}(\delta_f)|
\text{GRot} &\left[R_{s, \sigma}\right]_g = [\nu \bar{x}((\nu r)[\sigma]_g(r) \mid [\sigma]_g)]
\text{GCon} &\left[[A]\right]_g = \left[\{(s_1, \sigma_1) \ldots (s_n, \sigma_n)\}, \delta_c\right], \delta_f\left[\right]_g \\
&= (\nu c \bar{\chi})(\nu f)(R_{s_1, \sigma_1} \ldots R_{s_n, \sigma_n})[[\delta_f]_g], \delta_c
\end{align*}
\]

The encoding of a function declaration is a replication of input prefix process. Each replica starts by inputting the argument lists along $f$, followed by a composition of processes denoting function parameters and function body. Since function does not return anything and a normal function call is forbidden, the termination signal is worthless, therefore a local name $r'$ is used in the encoding of the function body.

$R_{s, \sigma}$ represents a Go-routine in which $s$ is to be executed with local store $\sigma$. Each $x$ in $\bar{x}$ refers to a local variable. In the encoding of a global configuration, the names referring to local channels $(\bar{c})$ and functions $(\bar{f})$ are local names.

The valid buffer store $\delta_c$ is obtained from the channel store $\delta_i$ in three steps: Firstly, prefix a $\nu$ symbol to the names referring to local channels in all buffers; Secondly, remove these unbuffered channels from the domain of $\delta_c$; And finally, for any buffered channel $\bar{c}h$ in the domain of $\delta_c$, eliminate the third element $g$ from $\delta_c(\bar{c}h)$.
3.3 Correctness

The correctness of the encoding is demonstrated by a full abstraction theorem with respect to (weak) bisimulation. The following lemma says that a global transition may be simulated by a nontrivial sequence of transitions of its encoding. Usually, the encoding will perform some internal adjustments before and after the real simulation.

**Lemma 15.** If $A \xrightarrow{\alpha} A'$, then $\llbracket A \rrbracket_g \Rightarrow \llbracket A' \rrbracket_g$

The lemma is proved by detailed analyze of the global transition rules one by one.

**Proposition 16.** Suppose the transition is inferred by global transition rule $\text{Loc}$, that is

$$\langle s, \sigma, \delta \rangle \xrightarrow{\ell_f} \langle s', \sigma', \delta' \rangle$$

then

$$(\nu \tilde{c} \chi)R_{s,\sigma,\delta} \Rightarrow \Rightarrow (\nu \tilde{c} \chi)R_{s',\sigma',\delta'}$$

where $\tilde{c}$ are local channels.

**Proof.** Consider the local transition rules which can be applied in the last step of the inference of premise.

Suppose the premise is an instance of local transition rule $\text{Ass}$ or $\text{SdB}$, the results follows by a detailed analyze on the actions of the encoding.

Suppose the premise is an instance of $\text{AsE}$, and the premise of this instance is an instance of local expression rule $\text{Var, Mak}$ or $\text{RvB}$, the results follows by a detailed analyze. If the premise of this instance is an instance of local expression rule $\text{RvE}$, we prove by induction on the depth of the inference of the premise of the instance. Suppose $s \equiv x = \leftarrow e$, $s' \equiv x = \leftarrow e'$ where $\langle e, \sigma, \delta' \rangle \xrightarrow{g} \langle e', \sigma', \delta'' \rangle$ by a shorter inference. By induction

$$(\nu \tilde{c} \chi)R_{s,\sigma,\delta} \Rightarrow \Rightarrow (\nu \tilde{c} \chi)R_{s',\sigma',\delta'}$$

From the definition of encoding for $x = e$ and $\leftarrow e$, it follows that

$$(\nu \tilde{c} \chi)R_{s,\sigma,\delta} \Rightarrow \Rightarrow (\nu \tilde{c} \chi)R_{s,\sigma,\delta}$$

$\text{SdE1, SdE2, GoE, SlR, or SlE}$ are similar to $\text{AsE}$.

Suppose the premise is an instance of $\text{Sls}$, $\text{Seq1}$ or $\text{Seq2}$, then we prove by induction on the depth of the inference of the premise. For $\text{Seq1}$ $s \equiv s_1; s_2$, $s' \equiv s'_1; s'_2$ where $\langle s_1, \sigma, \delta \rangle \xrightarrow{g} \langle s'_1, \sigma', \delta' \rangle$ by a shorter inference. By induction

$$(\nu \tilde{c} \chi)R_{s_1,\sigma,\delta} \Rightarrow \Rightarrow (\nu \tilde{c} \chi)R_{s'_1,\sigma',\delta'}$$

From the definition of encoding for $s_1; s_2$, it follows that

$$(\nu \tilde{c} \chi)R_{s,\sigma,\delta} \Rightarrow \Rightarrow (\nu \tilde{c} \chi)R_{s,\sigma,\delta}$$

$\text{Sls}$ and $\text{Seq2}$ are similar to $\text{Seq1}$.

This completes the proof. $\square$
Proposition 17. Suppose the transition is inferred by global transition rule \( L\text{Go} \), that is
\[
\delta_f \vdash (\Gamma \cup \{(s, \sigma), \delta_c\}) \rightarrow_g (\Gamma \cup \{(s', \sigma), (sf, \sigma_f), \delta_c\})
\]
where \( \delta_f(f) = (\text{func } f(x_1 \ldots x_m) \{ s_f \}) \) and \( \sigma_f = [x_1 \mapsto v_1 \ldots x_m \mapsto v_m] \), then
\[
(\nu \tilde{\chi})(R_{s, \sigma}(\delta_f^g)), \tilde{\delta}_c \Rightarrow \Rightarrow (\nu \tilde{\chi})(R_{s', \sigma}(R_{s_f, \sigma_f}(\delta_f^g)), \tilde{\delta}_c^g)
\]
where \( \tilde{\chi} \) are local channels.

Proof. Consider the local transition rules which can be applied in the last step of the inference of premise.

Suppose the premise is an instance of local transition rule \( \text{Go} \), the results follows by a detailed analyze on the actions of the encoding.

Suppose the premise is an instance of SeQ1 or SeQ2, then we prove by induction on the depth of the inference of the premise. For SeQ1 \( s \equiv s_1; s_2, s' \equiv s'_1; s_2 \) where \( (s_1, \sigma, \delta_c) \overset{\text{func } f(x_1 \ldots x_m) \{ s_f \}}{\longrightarrow}_g (s'_1, \sigma, \delta_c) \) by a shorter inference. By induction
\[
(\nu \tilde{\chi})(R_{s, \sigma}(\delta_f^g)), \tilde{\delta}_c \Rightarrow \Rightarrow (\nu \tilde{\chi})(R_{s'_1, \sigma}(R_{s_f, \sigma_f}(\delta_f^g)), \tilde{\delta}_c^g)
\]
From the definition of encoding for \( s_1; s_2 \), it follows that
\[
(\nu \tilde{\chi})(R_{s_1; s_2, \sigma}(\delta_f^g)), \tilde{\delta}_c \Rightarrow \Rightarrow (\nu \tilde{\chi})(R_{s'_1; s_2, \sigma}(R_{s_f, \sigma_f}(\delta_f^g)), \tilde{\delta}_c^g)
\]
SeQ2 are similar to SeQ1.

This completes the proof. \( \square \)

For other global transition, it is similar.

Conversely, a sequence of transitions of \([A]_g \) should reflect certain global transitions of \( \Lambda \). However it is not always possible, since the simulation may not yet complete, even worse the transition sequence simulating one global transition may interleave with transition sequences simulating others. Fortunately, by observing the proof of the previous lemma, we find that actually only one transition in the sequence plays the crucial role, as this transition uniquely identifies a global transition. Other \( \tau \) transitions, whether preceding or following this special transition, are internal adjustments which prepare for the special transition immediately after them. We call the special transition a simulating transition, and the other non-special \( \tau \) transitions preparing transitions.

Preparing transitions are local synchronous communication between sub-processes of one single Go-routine (e.g. synchronous communication making sub-processes evolve in order). To postpone or to advance preparing transitions would not affect the behavior of other Go-routines.

These observations are formulated by the following definitions and lemmas.

Definition 18. A transition \( P, B \overset{\alpha}{\rightarrow} P', B' \) is a simulating transition if the action \( \alpha \) is induced by the underlined prefixes and extended new operators specified in the encoding in Table 11. Otherwise, it is a preparing transition.
Definition 19. Let \( A \) be a global configuration, the set \( T_A \) is defined as follows:

1. \( \llbracket A \rrbracket_g \in T_A \)
2. If \( (P, B) \in T_A \) and \( (P, B) \rightarrow (P', B) \) is a preparing transition, then \( (P', B) \in T_A \)
3. If \( (P, B) \in T_A \) and \( (P', B) \rightarrow (P, B) \) is a preparing transition, then \( (P', B) \in T_A \)

Proposition 20. Let \( A \) be a global configuration, and \( \llbracket A \rrbracket_g = (\nu \hat{c}h)(\nu \hat{f}) \)

\[ (R_1 | \ldots | R_n | \llbracket \delta f \rrbracket_g), \hat{\delta} \] . Suppose \( \llbracket A \rrbracket_g = P, B \), then

\[ P \equiv_B (\nu \hat{c}h')(\nu \hat{c}h)(P_1 | \ldots | P_n | \ldots | P_m | \llbracket \delta f \rrbracket_g) \]

where \( P_i \) (i \( \leq \) n) is (subprocess of) a descendant of \( R_i \), \( P_j \) \( (j > n) \) corresponds to a newly created Go-routine. Also, \( \{\hat{c}h'\} \subset \{\hat{c}h\} \) and \( \forall \hat{c}h \in \{\hat{c}h\} \setminus \{\hat{c}h'\} \). \( \overrightarrow{d}(\nu \hat{c}h) \in t \), where \( d \) refers to a global channel when \( \overrightarrow{d}(\nu \hat{c}h) \) happens. Each name in \( \hat{c}h'' \) denotes newly created channels during \( \Rightarrow \).

Proposition 21. If \( P, B \rightarrow P', B' \) is a preparing transition, then

1. \( B = B' \)
2. It is a preparing transition of \( i \)-th Go-routine, i.e. \( P \) and \( P' \) differs only on \( P_i \) for some \( i \).
3. The transition is induced by \( P_i, B'' \rightarrow P'_i, B'' \) where \( B'' = B{\hat{c}h/\nu \hat{c}h} \).

Lemma 22. If \( T_A \ni (P, B) \rightarrow^* (P', B') \), and only \( \alpha' \rightarrow \) is a simulating transition, then there exists \( A' \) such that

\[ A \xrightarrow{g} A' \text{ and } (P', B') \in T_A \]

Proof. By the Definition 18 and Propositions 20 and 21. \( \square \)

Any of the processes in \( T_A \) can be seen as the encoding of \( A \).

Lemma 23. If \( (P, B) \in T_A \) and \( (Q, B) \in T_A \), then we have \( (P, B) \approx (Q, B) \).

Proof. The following relation is a bisimulation.

\[ R = \{((P, B), (Q, B)) \mid (P, B) \in T_A \land (Q, B) \in T_A \} \]

\( (P, B) \rightarrow (P', B') \in T_A \) is trivial.

Suppose \( (P, B) \rightarrow (P', B') \notin T_A \) is a simulating transition. By Lemma 22, there exists \( A' \) such that

\[ A \xrightarrow{g} A' \text{ and } (P', B') \in T_A \]
Suppose this $\alpha \rightarrow g$ involves the $i$-th (and $j$-th) Go-routine. For $(Q, B)$, perform the preparing transitions of the $i$-th (and $j$-th) Go-routine, followed by $\alpha' \rightarrow$, we have

$$(Q, B) \Rightarrow^{\alpha'} (Q', B') \in T_{\Lambda'}$$

The other direction is similar. $\square$

As a consequence, bisimulation is preserved by the encoding.

**Theorem 24.** $A_1 \approx_g A_2$ if and only if $\llbracket A_1 \rrbracket_g \approx \llbracket A_2 \rrbracket_g$.

**Proof.** $\Rightarrow$: The following relation is a bisimulation up to $\approx$.

$$\mathcal{R} = \{ ([A_1]_g, [A_2]_g) \mid A_1 \approx_g A_2 \}$$

Suppose $\llbracket A_1 \rrbracket_g \overset{\alpha'}{\rightarrow} (P, B_P) \notin T_{A_1}$ is a simulating transition. By Lemma 22 and 23, there exists $A'_1$ such that

$A_1 \overset{\alpha}{\rightarrow}_g A'_1$ and $T_{A'_1} \ni (P, B_P) \approx \llbracket A'_1 \rrbracket_g$

Since $A_1 \approx_g A_2$, there exist $A'_2$ such that

$A_2 \overset{\hat{\alpha}}{\Rightarrow}_g A'_2 \approx_g A'_1$

By Lemma 15

$\llbracket A_2 \rrbracket_g \overset{\hat{\alpha}'}{\Rightarrow} \llbracket A'_2 \rrbracket_g \in T_{A'_2}$

The other direction is similar.

$\Leftarrow$: The following relation is a bisimulation.

$$\mathcal{R} = \{ (A_1, A_2) \mid [A_1]_g \approx [A_2]_g \}$$

Suppose $A_1 \overset{\alpha}{\rightarrow}_g A'_1$, by Lemma 15

$\llbracket A_1 \rrbracket_g \Rightarrow^{\alpha'} \llbracket A'_1 \rrbracket_g$

Since $[A_1]_g \approx [A_2]_g$, there exist $(Q, B_Q)$ such that

$\llbracket A_2 \rrbracket_g \overset{\hat{\alpha}'}{\Rightarrow} (Q, B_Q) \approx \llbracket A'_1 \rrbracket_g$

For each simulating transitions of $\overset{\hat{\alpha}'}{\Rightarrow}$, by Lemma 22, there exists $A'_2$, such that

$A_2 \overset{\hat{\alpha}}{\Rightarrow}_g A'_2$ and $T_{A'_2} \ni (Q, B_Q) \approx \llbracket A'_2 \rrbracket_g$

The other direction is similar. $\square$
Functions:

\[ f ::= \text{fun} \ (x_1 \ldots x_n) \Rightarrow e \]

Expressions:

\[ e, e_1, e_2, \ldots ::= n \quad \text{Integer} \]
\[ \mid id \quad \text{Process id} \]
\[ \mid x \quad \text{Variable} \]
\[ \mid \text{let } x = e_1 \text{ in } e_2 \quad \text{Let binding} \]
\[ \mid \text{apply } f(n) \ (e_1 \ldots e_n) \quad \text{Function application} \]
\[ \mid \text{spawn } fn \ [e_1 \ldots e_n] \quad \text{Spawn Erlang Process} \]
\[ \mid e_1 \ ! \ e_2 \quad \text{Sending} \]
\[ \mid \text{receive } c_1 \ldots c_n \quad \text{receiving} \]

where \( c_1, c_2, \ldots ::= x \text{ when } e_1 \Rightarrow e_2 \)

Table 13. Syntax of Core Erlang

4 Core Erlang

We improve the translation mapping showed in [12] by a fully abstract encoding in the \( \pi_b \)-calculus.

4.1 Syntax of Core Erlang

The syntax of a subset of Core Erlang is presented in Table 13.

\texttt{let} binds values to variables, and functions are bound to function names by function definitions in the form

\[ fn = f \]

The counterpart of Go-routine in Erlang is the \textit{Erlang process}. Each Erlang process is identified by an unique process id. Moreover every Erlang process is associated with a mailbox which is an unbounded ordered sequence. The Erlang process is created by the \texttt{spawn} expression. This expression acts almost the same as the \texttt{go} statement except that it is an expression and takes the newly created Erlang process’s \texttt{id} as result.

Communication in Erlang is asynchronous. Send expression \( e_1 \ ! \ e_2 \) appends message \( e_2 \), which is also the result of this expression, to the mailbox of the Erlang process identified by \( e_1 \). Receive operation is based on pattern matching. The receive clause is deliberately simplified to "\( x \text{ when } e_1 \Rightarrow e_2 \)" , where \( x \) is an “always march” pattern. Once a receive expression occurs, messages reside in the mailbox of the Erlang process evaluating this expression are tried in first-to-last order against the clauses \( c_1 \ldots c_n \) from left to right. For message \( v \) and clause \( c \), pattern marching results in \( x \) binding to \( v \) in \( e_1 \) and \( e_2 \). If the guard expression \( e_1 \) evaluates to the Erlang atom ‘\texttt{true}’, matching succeeds, message \( v \) is deleted (received) from the mailbox, and the result of the expression is the result of \( e_2 \). Otherwise, the next clause will be tried by \( v \). If no more clause left for \( v \), that is \( v \) does not match any clause, then the next message in mailbox
will be used for marching, with \( v \) remains in the mailbox. Sometimes none of the existing messages matches any clause, in this case receive blocks until new message arrives.

Without loss of generality, we stipulate a Core Erlang program is a set of function definitions, in which a function named \( \text{start} \) is defined. Running a program is equivalent to evaluate \( \text{spawn} \ \text{start} \ [\ldots] \) with appropriate arguments. For the sake of simplicity, we assume the function bodies contain no local variables other than function arguments. Note that function name may appear only at the function position of spawn or apply expressions in this subset language — high-order is not considered.

### 4.2 Operational Semantics

The structural operational semantics of Core Erlang is also defined by a two-level labelled transition system: the local transition system specifies the evaluation of a single Erlang process in isolation, and the global transition system describes the behavior of a running Erlang program.

The local transition system defines transition rules between local configurations. A local configuration is a tuple \( (e, m) \) where \( e \) is the expression to be evaluated by Erlang process whose mailbox is \( m \). The local transition rules, defined in Table 14, are of the form

\[
\delta_f, id \vdash (e, m) \xrightarrow{\alpha} (e', m')
\]

where \( \delta_f \) is a mapping from function names to functions, and \( id \) identifies the Erlang process evaluating the expression. Actions can be either silent action \( \tau \), \( \text{sd}(id, v) \) denoting send action, or \( \text{sp}(fn, v_1 \ldots v_n) \) denoting Erlang process creation.

Subexpression evaluation in Core Erlang is strict, however, in which order a sequence of subexpressions are evaluated is not defined. This evaluation strategy is specified by \( \text{LtE}, \text{ApE}, \text{SpE}, \text{SdE1} \) and \( \text{SdE2} \).

**LET** and **APP** is straightforward. In **SPA**, the **sp** label indicates that the new Erlang process is identified by \( id' \) and the expression it will evaluate is the function application \( \text{apply} \ fn(\bar{v}) \). Sending a message to an Erlang process carries the sending label \( \text{sd}(id, v) \), while receiving is silent. The premise of **Rcv** indicates that the first suitable message is the \( k \)-th message, and it marches the \( i \)-th clause.

Global transitions happen between global configurations which contain information of all running Erlang processes. A global configuration, denoted by \( \Lambda, A_1, \ldots, \) is defined as a tuple \( (ID, E, \delta_m) \), where \( ID \) and \( E \) are the sets of \( ids \) and expressions, respectively, of all running Erlang processes, and \( \delta_m \) is the mailbox store. A mailbox store \( \delta_m \) is a mapping from process \( ids \) to pairs \( (m, g) \), where \( m \) is a mailbox (a list) and \( g \) is a tag indicating whether (the mailbox of) the Erlang process is accessible by an observer (1) or not (0).

We say an Erlang process is local if it is created during the evaluation of a program. The set of local Erlang process \( ids \) is exactly the \( ID \) of a global
Let $\delta_f, id \vdash \langle \text{let } x = v \text{ in } e_2, m \rangle \rightsquigarrow e_2(v/x), m$

\[ \delta_f, id \vdash \langle e_1, m \rangle \xrightarrow{\alpha} e_1', m' \]

LTe

\[ \delta_f, id \vdash \langle \text{let } x = e_1 \text{ in } e_2, m \rangle \xrightarrow{\alpha} e \]

\[ \delta_f(id) = \text{fun}(\overline{x}) \Rightarrow e \]

App

\[ \delta_f, id \vdash \langle \text{apply } fn(\overline{x}), m \rangle \xrightarrow{\alpha} e \]

\[ \delta_f(id) \vdash \langle e_1, m \rangle \xrightarrow{\alpha} e_1', m' \}; \ i \in [n] \]

APE

\[ \delta_f, id \vdash \langle \text{apply } fn(e_1), m \rangle \xrightarrow{\alpha} e \]

\[ \text{id}' \text{ is fresh} \]

Spa

\[ \delta_f, id \vdash \langle \text{spawn } fn \overline{x}, m \rangle \xrightarrow{\text{sp}(\text{id}', \overline{x}, \overline{\overline{x}})} \text{id}' \]

SpE

\[ \delta_f, id \vdash \langle \text{spawn } fn e_1, m \rangle \xrightarrow{\alpha} e \]

\[ \delta_f(id) \vdash \langle e_1, m \rangle \xrightarrow{\alpha} e_1', m' \}; \ i \in [n] \]

Snd

\[ \delta_f, id \vdash \langle \text{id}'! v, m \rangle \xrightarrow{\text{id}(\text{id}', v)} \langle v, m \rangle \]

SdE1

\[ \delta_f, id \vdash \langle e_1, m \rangle \xrightarrow{\alpha} e_1', m' \]

SdE2

\[ \delta_f, id \vdash \langle e_1 \leftarrow e_2, m \rangle \xrightarrow{\alpha} e_1', e_2', m' \]

\[ \forall 0 < l < k \land 0 < j \leq n \]

\[ \delta_f(id) \vdash \langle e_1'[m/l/x'], m \rangle \xrightarrow{\alpha} \langle \text{'false'}, m \rangle \]

\[ \forall 0 < j < i \]

\[ \delta_f(id) \vdash \langle e_1'[m[k/x'], m \rangle \xrightarrow{\alpha} \langle \text{'false'}, m \rangle \]

\[ \delta_f(id) \vdash \langle e_1'[m[k/x'], m \rangle \xrightarrow{\alpha} \langle \text{'true'}, m \rangle \]

Rcv

\[ \delta_f, id \vdash \langle \text{receive } e_1, \ldots, e_n, m \rangle \xrightarrow{\alpha} e_2'[m[k/x'], m[k]] \]

\[ \text{where } e_j \equiv x \text{ when } e_1' \Rightarrow e_2' \text{ and } m[k] \]

\[ \text{means remove the } k\text{-th element from } m \]

Table 14. Local Transition Rules of Core Erlang
configuration. All the Erlang processes in the environment (context) are global. An local Erlang process is accessible if the environment knows its process id.

A global transition takes the form

$$\delta_f \vdash \langle ID_1, E_2, \delta_{m_1} \rangle \overset{\alpha}{\rightarrow} \langle ID_2, E_2, \delta_{m_2} \rangle$$

An Erlang program will start from an initial configuration

$$\langle \{id_{start}\}, \{e_{start}\}, \delta_{init} \rangle$$

where id_{start} is a fresh process id, e_{start} is the expression obtained from the body of start by simultaneously substituting supplied arguments for parameters of the function, and $$\delta_{init} = [id_{start} \mapsto ([], 0)]$$ is the initial mailbox store. The global transition rules are listed in Table 14.2. A global action can be either $$\tau$$, s(id, v) for sending, or r(id, v) for receiving.

$$\begin{align*}
\text{Loc} & : \delta_f, id \vdash \langle e, m \rangle \mapsto \langle e', m' \rangle \\
\text{LSp} & : \delta_f, id \vdash \langle e, m \rangle \xrightarrow{sp(id', v)} \langle e', m' \rangle; \ id' \notin ID; \ \delta_f(f) = \text{fun} (\bar{x}) \Rightarrow e' \\
\text{LSD} & : \delta_f, id \vdash \langle e, m \rangle \xrightarrow{id(id', m)} \langle e', m' \rangle; \ id' \in ID; \ \delta_m(id') = (m', g') \\
\text{GSp1} & : \delta_f, id \vdash \langle e, m \rangle \xrightarrow{id(id', v)} \langle e', m' \rangle; \ id' \notin ID; \ v \notin ID \\
\text{GSp2} & : \delta_f, id \vdash \langle e, m \rangle \xrightarrow{id(id', id'')} \langle e', m' \rangle; \ id' \notin ID; \ id'' \in ID; \ \delta_m(id'') = (m', g') \\
\text{GRV} & : \delta_f \vdash \langle ID, E, \delta_m \rangle \xrightarrow{r(id,v)} \langle ID, E, \delta_m[ id \mapsto (m :: [v], 1)] \rangle
\end{align*}$$

Table 15. Global Transition Rules of Core Erlang

Loc specifies the independent evaluation of an Erlang process. Receive operation will also take this transition since Rcv is a silent local transition. LSp creates a new Erlang process. LSD defines the sending operation between two local Erlang processes. The rules Loc, LSp and LSD all specify internal actions of a running program.

The labels in GSp1, GSp2 and GRV indicate how an Erlang program interacts with the environment. An Erlang program can send values to the environment via global Erlang process id's, this behavior is captured by GSp1 and GSp2. The latter also describe how an inaccessible Erlang process becomes an accessible
one. Note that the $\nu$ symbol in the label $s(id', \nu id'')$ in $\text{GSd}2$ is required only when $id''$ denotes an unaccessible Erlang process, i.e., $g' = 0$. The environment can also send values to an accessible Erlang program via its ids. In $\text{GRv}$ the label $r(id, v)$ actually means that the accessible Erlang process id “receives” a value $v$ from the environment.

**Definition 25.** A symmetric binary relation $R$ over global configurations is a (weak) bisimulation if

$$A_1 R A_2 \land A_1 \xrightarrow{\alpha} A'_1 \text{ then } \exists A'_2 . A_2 \xrightarrow{\hat{\alpha}} A'_2 \land A'_1 R A'_2$$

Two global configurations are bisimilar, written as $A_1 \approx_e A_2$, if they are related by some bisimulation.

Two Erlang programs $ep_1, ep_2$ are bisimilar, if their initial global configurations (with the same $\delta_m$) are bisimilar.

### 4.3 Encoding of Core Erlang

The encoding of Core Erlang in $\pi_b$ calculus is achieved by the translation function $[e]_e(a, p, r)$. This function, defined in Table 16, takes three parameter: the first parameter $a$ stands for the id and the “input port” of the mailbox of the Erlang process evaluating $e$; the “output port” of the mailbox is obtained from the second parameter $p$; and the result of $e$ is returned along the last parameter $r$.

In $\text{Spaw}$, the input port (also the process id) and the output port of the new Erlang process’s mailbox is $a'$ and $b'$ respectively. Result of the function application is returned via local name $r'$ and hence simply dropped.

**Receive** We use the following algorithm to simulate one receive operation.

The algorithm uses two buffered names for each receive — a newly created buffered name and the buffered name created by a previously receive. From the viewpoint of a receiver, the latter is the output port of the mailbox from which messages are retrieved. Once the receive operation succeeds, the former will become the output port of the mailbox. The encoding of receive expression is basically the implementation of the algorithm in the $\pi_b$-calculus.

In $\text{Recv}$, the previous buffered name, say $b$, is saved in the second parameter of the translation function ($p$). After creating a new buffered name $b'$, process $RH$ is triggered. Receive handle process $RH$ fetches a message from mailbox $b$ and passes it to the first clause process for matching. Clause process $RC_e(s, s')$ gets the message from its first parameter. If guard expression evaluates to ‘true’, matching succeeds. The corresponding clause body process begins its evaluation with the previous buffered name changed to $b'$ and a copying process $Cp$ carries all remaining messages from the old mailbox to the new one. Otherwise, the message is passed along the second parameter to the next clause process for matching. If the message does not match the last clause, it is passed back to $RH$ which then inserts the message to the new mailbox $b'$ and starts the matching
4. Set the previous buffer to \( b' \). Create a new buffered name \( b \).

5. Retrieve a message \( v \) from \( b \), pattern match \( v \) against the first clause.

6. Pattern match \( v \) against clause \((x \text{ when } e_1 \rightarrow e_2)\), substitute \( v \) for each free \( x \) in \( e_1 \) and \( e_2 \). If \( e_1 \) evaluates to ‘true’, goto 4; otherwise pattern match \( v \) against the next clause, goto 4. If no clause remains, insert \( v \) into \( b' \) and goto 4.

7. Set the previous buffer to \( b' \), copy all remaining messages from \( b \) to \( b' \), evaluate \( e_2 \).
of the next message. In the encoding, $RC$ is guarded by its first parameter $s$, and all $RC$s are chained together by local names $s_1 \ldots s_{n+1}$. A clause process cannot proceed unless the matching of the previous clause failed.

In general, we have the following proposition concerning the mailbox.

**Proposition 26.** A mailbox is explicitly modelled as follows

$$M(a, b^n) = Cp(a, b)|Cp(b, b^1)| \ldots |Cp(b^{n-1}, b^n)$$

where $a$ and $b^n$ are input and output ports of the mailbox. Each buffered $b^j$ ($j > 1$) is created by one receive operation, and the first buffered $b$ is created by the spawn expression. Send expressions insert messages into the mailbox via input port $a$, while receive expressions retrieve messages from the mailbox via output port $b^n$.

**Configuration** The translation function can be extended to a mapping from global configurations (with $\delta_f$) to $\pi_b$ processes. We write $[[\Lambda]]_e$ for the pair $(P, B)$, where $P$ is the encoding of $\Lambda$ and $\delta_f$, and $B$ is a valid buffer store inferred from mailbox store $\delta_m$. The extended translation function is shown in Table 18.

| FDef | $[\text{fn} = \text{fun } (x_1 \ldots x_n) \rightarrow e]_e = !\text{fn}(a, p, r, x_1 \ldots x_n).[[e]_e(a, p, r)$ |
| FEnv | $[[\delta_f]]_e = \prod_{i=1}^{n}[[\delta_f(f_i)]]_e \quad f_i \in \text{dom}(\delta_f); n = |\text{dom}(\delta_f)|$ |
| EPro | $R_{e, m}(a) = (vpr)(vbb^1 \ldots b^k)([e]_e(a, p, r) \mid \mathcal{P}(b^k) \mid M(a, b^k))$ |
| ECon | $[\Lambda]_e = [[ID, E, \delta_m], \delta_f]_e = (v\tilde{a})(v\tilde{fn})(R_{e_1, m_1}(a_1) \ldots |R_{e_n, m_n}(a_n)||[\delta_f]_e), \delta_m$ |

Table 18. Extended $[\cdot]_e$

The encoding of a function definition is a replication of input prefix process. Each replica starts by inputting the argument lists along $\text{fn}$, followed by the processes denoting the function body.

$R_{e,m}(a)$ represents an Erlang Process $(a)$ ready to evaluate expression $e$ with mailbox $m$ whose input port is $a$. In the encoding of a global configuration, the names referring to local Erlang Processes which are not accessible $(\tilde{a})$ and functions $(\tilde{fn})$ are local names.

#### 4.4 Correctness

The correctness of the encoding can be demonstrated by a similar analyze as Go. The following lemma says substitution for free variables can be postponed after the encoding.

**Lemma 27.**

$$[e\{x/v\}]_e(a, p, r) = ([e]_e(a, p, r))\{x/v\}$$
Proof. Simple induction on the structure of $e$.

**Theorem 28.** If $\delta_f \vdash A \xrightarrow{\alpha} A'$, then $[A]_e \Rightarrow [A']_e$.

**Proposition 29.** Suppose the transition is inferred by global transition rule Loc, that is 

$$
\delta_f, id \vdash \langle e, m \rangle \Rightarrow \langle e', m' \rangle
$$

then

$$(\nu a)R_{e, m}(a), \delta_m \Rightarrow \Rightarrow (\nu a)R_{e', m'}(a), \delta'_m$$

where $\alpha$ are local non-accessible Erlang Processes, and $\delta'_m = \delta_m[id \mapsto (m', g)]$.

**Proof.** We prove by simultaneous induction on the depth of inference of the premises. Consider the local transition rules applied in the last step of the inference of premise.

For $\text{Recv}$: Suppose $e \equiv \text{receive } c_1 \ldots c_n; e' \equiv e'_1[m[k]/x']$; $m' = m/[k]$, where

$$
\forall 0 < l < k \land 0 < j \leq n. \delta_f, id \vdash \langle e'_1[m[l]/x'], m \rangle \xrightarrow{e} ('false', m)
$$

$$
\forall 0 < j < i. \delta_f, id \vdash \langle e'_1[m[k]/x'], m \rangle \xrightarrow{e} ('false', m)
$$

$\delta_f, id \vdash \langle e'_1[m[k]/x'], m \rangle \xrightarrow{e} ('true', m)$

by a shorter inference. By induction

$$
\forall 0 < l < k \land 0 < j \leq n. (\nu a)R_{e'_1[m[l]/x'], m}(a), \mathcal{B} \Rightarrow \Rightarrow (\nu a)R_{false, m}(a), \mathcal{B}
$$

$$
\forall 0 < j < i. (\nu a)R_{e'_1[m[k]/x'], m}(a), \mathcal{B} \Rightarrow \Rightarrow (\nu a)R_{false, m}(a), \mathcal{B}
$$

$$(\nu a)R_{e'_1[m[k]/x'], m}(a), \mathcal{B} \Rightarrow \Rightarrow (\nu a)R_{true, m}(a), \mathcal{B}
$$

From the definition of encoding for $\text{receive } c_1 \ldots c_n$, it follows that

$$(\nu a)R_{e, m}(a), \delta'_m = (vpr)(\nu bbb \ldots b^k)([\text{receive } c_1 \ldots c_n]_c(a, p, r)\mathcal{P}[b^k], M(a, b^k)), \delta'_m
$$

$$
\Rightarrow (vpr)(\nu bbb \ldots b^k+1)
$$

$$
([e'_1[m[k]/x']])_c(a, p, r)\mathcal{P}[b^{k+1}], M(a, b^{k+1})[\text{Cp}(b^{k}, b^{k+1})], \delta'_m
$$

$= R_{e', m'}(a), \delta'_m$

Other cases are similar. $$\square$$

**Proposition 30.** Suppose the transition is inferred by global transition rule LSp, that is 

$$
\delta_f, id \vdash \langle e, m \rangle \xrightarrow{\text{sp}(id', fn, \delta)} \langle e'', m \rangle; \text{id}' \notin \text{ID}
$$

where $\delta_f(fn) = \text{fun} (\tilde{x}) \Rightarrow e'$ and $\delta'_m = \delta_m[id' \mapsto ([\tilde{x}], 0)]$, then

$$(\nu a)(R_{e, m}(a))[\delta_f]_e, \delta'_m \Rightarrow \Rightarrow (\nu a'R_{e', m}(a))R_{e'[\tilde{x}], [\tilde{a}'][\delta_f]_e}, \delta'_m$$
For other global transition, it is similar.

Although argument evaluation is strict in Core Erlang, the evaluation order of a sequence of argument expressions is undefined. In the encoding, besides the interleaving between transitions of many Erlang processes, interleaving also exists inside one single Erlang process — between the transitions simulating argument evaluation. This interleaving is not serious, since except for receive and send, expression evaluation has no side effect, the transitions of one argument process will not affect the behavior of others. For receive, the input prefix \( p(b) \) also acts as a semaphore which prevents two receive operations run in parallel. For send, it may only affect the behavior of receive. But according to the operational semantics, receive expression would proceed unless a legal message is already in the mailbox. Hence the interleaving of transition simulating send and receive expression can be rearranged in a non-interleaving way.

**Definition 31.** A transition \( P, B \xrightarrow{\alpha} P', B' \) is a simulating transition if the action \( \alpha \) is induced by underlined prefixes specified in Table 16. Otherwise, it is a preparing transition. For receive, only the transition \( \tau \xrightarrow{\text{true}} \) is a simulating transition.

**Definition 32.** Let \( A \) be a global configuration, the set \( \mathcal{T}_A \) is defined as follows:

1. \( \llbracket A \rrbracket_e \in \mathcal{T}_A \)
2. If \( (P, B) \in \mathcal{T}_A \) and \( (P, B) \xrightarrow{\mathcal{T}} (P', B') \) is a preparing transition, then \( (P', B') \in \mathcal{T}_A \)

**Lemma 33.** If \( \mathcal{T}_A \ni (P, B) \Rightarrow \xrightarrow{\tau} (P', B') \), and only \( \xrightarrow{\tau} \) is a simulating transition, then there exists \( A' \) such that

\[ A \xrightarrow{\alpha_e} A' \quad \text{and} \quad (P', B') \in \mathcal{T}_A \]

Any of the processes in \( \mathcal{T}_A \) can be seen as the encoding of \( A \).

**Lemma 34.** If \( (P, B) \in \mathcal{T}_A \) and \( (Q, B') \in \mathcal{T}_A \), then we have \( (P, B) \approx_e (Q, B') \)

As a consequence, bisimulation is preserved by the encoding.

**Theorem 35.** \( A_1 \approx_e A_2 \) if and only if \( \llbracket A_1 \rrbracket_e \approx \llbracket A_2 \rrbracket_e \)

5 Conclusion and Future Work

We have presented the \( \pi_b \)-calculus which extends the \( \pi \)-calculus by buffered names. Communication along buffered names is asynchronous, i.e. native support of asynchronous communication. After presenting its syntax and semantics, we give out a full abstract encoding of the \( \pi_b \) calculus in the traditional payadic \( \pi \)-calculus with respect to strong bisimulation. It is obvious that the new calculus
does not increase the expressive power. However, in contrast to the \( \pi \)-calculus which is hard to use in practice, it enables easy and clear modeling of practical concurrent languages. Specifically, we have provided encodings of two real-world concurrent languages — the (core) Go language and the Core Erlang — in the buffered \( \pi \)-calculus. Both encodings are fully abstract with respect to weak bisimulations.

The transition rules of the \( \pi_b \)-calculus are a bit complicated compared with that of the \( \pi \)-calculus. We aim at applying the new language for modeling and verifying large distributed and concurrent systems with asynchronous message passing-like communication by automatic computer programs. One line of future work is to develop such programs. We may extend existing tools such as Pict [13], MWB [16] or the HD Automata Laboratory [5] to handle the \( \pi_b \)-Calculus.

Since weak bisimulation is not sufficient to demonstrate program equivalence, we may expect some full abstraction encodings with respect to branching bisimulation, or even strong bisimulation.

Acknowledgement

The authors are partially supported by Natural Science Foundation of China (61173033, 61033002, 61100053). They would like to thank the members of BASICS for their interest in this work. They are also grateful to the three anonymous referees for their detailed comments on the previous version of the paper.

References

1. Armstrong, J.L.: The Development of Erlang. In: ICFP. pp. 196–203 (1997)
2. Beauxis, R., Palamidessi, C., Valencia, F.D.: On the Asynchronous Nature of the Asynchronous \( \pi \)-Calculus. In: Concurrency, Graphs and Models. pp. 473–492 (2008)
3. Boudol, G.: Asynchrony and the \( \pi \)-calculus. Rapport de recherche RR-1702, INRIA (1992), [http://hal.inria.fr/inria-00076939](http://hal.inria.fr/inria-00076939)
4. Carlsson, R.: An introduction to Core Erlang. In: PLI’01 Erlang Workshop (2001)
5. Ferrari, G.L., Gnesi, S., Montanari, U., Pistore, M.: A model-checking verification environment for mobile processes. ACM Trans. Softw. Eng. Methodol. 12(4), 440–473 (2003)
6. Google Inc.: The Go Programming Language Specification (2012), [http://golang.org/ref/spec](http://golang.org/ref/spec)
7. Hoare, C.A.R.: Communicating Sequential Processes. Communications of the ACM 21(8), 666–677 (1978)
8. Honda, K., Tokoro, M.: An Object Calculus for Asynchronous Communication. In: ECOOP. pp. 133–147 (1991)
9. Milner, R.: Communication and Concurrency. PHI Series in computer science, Prentice Hall (1989)
10. Milner, R.: The Polyadic \( \pi \)-Calculus: a tutorial. Tech. Rep. ECS-LFCS-91-180, Computer Science Dept., University of Edinburgh (1991)
11. Milner, R., Parrow, J., Walker, D.: A Calculus of Mobile Processes, Parts I and II. Information and Computation 100(1), 1–77 (1992)
12. Noll, T., Roy, C.K.: Modeling Erlang in the π-calculus. In: Erlang Workshop. pp. 72–77 (2005)
13. Pierce, B.C., Turner, D.N.: Pict: a programming language based on the pi-calculus. In: Proof, Language, and Interaction. pp. 455–494 (2000)
14. Roy, C.K., Noll, T., Roy, B., Cordy, J.R.: Towards Automatic Verification of Erlang Programs by π-Calculus Translation. In: Erlang Workshop. pp. 38–50 (2006)
15. Sangiorgi, D., Walker, D.: The π-Calculus: A Theory of Mobile Processes. Cambridge University Press (2001)
16. Victor, B., Moller, F.: The Mobility Workbench — a tool for the π-calculus. In: Dill, D. (ed.) CAV’94: Computer Aided Verification. Lecture Notes in Computer Science, vol. 818, pp. 428–440. Springer-Verlag (1994)
17. Walker, D.: Objects in the π-Calculus. Information and Computation 116(2), 253–271 (1995)
A Proofs

Proof of Lemma

Proof. We prove by induction on the depth of inference tree of the condition. Consider each rule in Table 2

1U, OQ, IB, OB, IBG, OBG and NewB* are the base step.

1U \( P = a(x).P''; \alpha = a(d); P' = P''[d/x]; B' = B \)

\[ [P, B] = a_1(x_1, x_2).[[P'']] \frac{a_1(d_1, d_2)}{d_1, d_2/x_1, x_2} [P''[d_1, d_2/x_1, x_2]][[B]] = [[P', B]] \]

OQ \( P = \overline{a}(d).P'; \alpha = \overline{a}(d); B' = B \)

\[ [P, B] = \overline{a}_2(d_1, d_2).[[P']][[B]] \frac{\overline{a}_2(d_1, d_2)}{d_1, d_2} [P'][[B]] = [[P', B]] \]

IB \( P = b(x).P''; \alpha = \tau; P' = P''[d/x]; B(b) = (n, [d] :: l); B' = B[b \mapsto (n, l)] \)

\[ [P, B] = b_1(x_1, x_2).[[P'']][F_n, [d_1, d_2]; L_b(b_1, b_2)][[B']][[B]] = [[P', B']] \]

OB \( P = \overline{b}(d).P'; \alpha = \tau; B(b) = (n, l); B' = B[b \mapsto (n, l :: [d])] \)

\[ [P, B] = \overline{b}_2(d_1, d_2).[[P'']][F_n, L_b(b_1, b_2)][[B']][[B]] = [[P', B']] \]

IBG \( P' = P; \alpha = b(d); B(b) = (n, l); B' = B[b \mapsto (n, l :: [d])]; \]

\[ [P, B] = (\nu \overline{c} \overline{c}_2)([[Q]] \quad [\Box \overline{c} \nu \overline{c}_2 \backslash b] \quad F_n, L_b(b_1, b_2)) = [[P', B]] \]

OBG \( P' = P; \alpha = \overline{b}(d); B(b) = (n, l); B' = B[b \mapsto (n, l)]; \]

\[ [P, B] = (\nu \overline{c} \overline{c}_2)([[Q]] \quad [\Box \overline{c} \nu \overline{c}_2 \backslash b] \quad F_n, L_b(b_1, b_2)) = [[P', B]] \]

NewB* \( P = (\nu b : n)P_1; P' = (\nu b)P_1; \alpha = \tau; B' = B[b \mapsto (n, l)] \)

\[ [P, B] = (\nu \overline{c} \overline{c}_2)([[Q]] \quad [\Box \overline{c} \nu \overline{c}_2 \backslash b] \quad F_n, L_b(b_1, b_2)) = [[P', B]] \]

where \( \{ \overline{c} \} = \ln (B) \cup \dom (B) \cap \ln (P) \), \( P \equiv_B (\nu \overline{c})Q \). Since \( b \in \dom (B) \) and \( b \notin \ln (P) \), then \( b \notin \{ \overline{c} \} \)

\[ [P, B] = (\nu \overline{c} \overline{c}_2)([[Q]] \quad [\Box \overline{c} \nu \overline{c}_2 \backslash b] \quad F_n, L_b(b_1, b_2)) = [[P', B]] \]

where \( \{ \overline{c} \} = \ln (B) \cup \dom (B) \cap \ln (P) \), \( P \equiv_B (\nu \overline{c})Q \) and \( Q = (\nu b : n)Q' \). Since \( b \notin \dom (B) \), then \( b \notin n(B) \)

\[ [P, B] = (\nu \overline{c} \overline{c}_2)(\nu b_2)[[Q']] \quad F_n, L_b(b_1, b_2) \quad [\Box \overline{c} \nu \overline{c}_2 \backslash b] = [[P', B[b \mapsto (n, l)]]] \]
\[ \text{Sum } P = \sum_{i \in I} \pi_i \cdot P_i; \pi_j \cdot P_j, B \xrightarrow{\alpha} P', B'. \] By induction,
\[ [\pi_j \cdot P_j, B] = [\pi_j, P_j] \mid [B] \xrightarrow{M(\alpha)} [P', B'] \]
\[ [P, B] = \sum_{i \in I} [\pi_i, P_i] \mid [B] \xrightarrow{M(\alpha)} [P', B'] \]

\[ \text{Par } P = P_1 \mid P_2; P_1, B \xrightarrow{\alpha} P'_1, B'; P' = P'_1 \mid P_2. \] By induction
\[ [P_1, B] = [P_1] \mid [B] \xrightarrow{M(\alpha)} [P'_1] \mid [P_2] \mid [B'] = [P'_1] \mid [P_2, B'] \]

\[ \text{Com } P = P_1 \mid P_2; P_1, B \xrightarrow{\pi(c)} P'_1, B; P_2, B \xrightarrow{\pi(c)} P'_2, B; P' = P'_1 \mid P'_2, B' = B. \] By induction,
\[ [P_1, B] = [P_1] \mid [B] \xrightarrow{\alpha_1, c_2} [P'_1, B] = [P'_1] \mid [B] \]
\[ [P_2, B] = [P_2] \mid [B] \xrightarrow{\pi(c)} [P'_2, B] = [P'_2] \mid [B] \]

Hence
\[ [P, B] = [P_1] \mid [P_2] \mid [B] \xrightarrow{\alpha} [P'_1] \mid [P'_2] \mid [B] = [P'_1] \mid [P'_2, B] \]

\[ \text{New* } P = (\nu c) P_1; P_1, B(c/\nu c) \xrightarrow{\alpha} P_2, B'; P' = (\nu c) P_2; B' = B'(\nu c/c). \] Suppose \( c \notin \text{ln}(B) \cup (\text{dom}(B) \cap \text{ln}(P)) \), the encoding is as follows
\[ [P, B] = (\nu d_1 d_2) [Q, B(d/\nu d)] = (\nu d_1 d_2)((\nu c_1 d_2) [Q'] \mid [B(d/\nu d)]) \]
where \( \{d\} = \text{ln}(B), P \equiv_B (\nu d) Q \) and \( Q = (\nu c) Q' \). Since \( c \notin \text{gn}(B) \), we can move the \( (\nu c_1 d_2) \) to the outermost level.
\[ [P, B] = (\nu c_1 d_2)(\nu d_1 d_2) ([Q'] \mid [B(d/\nu d)]) = (\nu c_1, c_2) [P_1, B(c/\nu c)] \]

By induction,
\[ [P_1, B(c/\nu c)] \xrightarrow{M(\alpha)} [P_2, B''] \]
\( c_1, c_2 \notin n(M(\alpha)) \), since \( c \notin n(\alpha) \), then
\[ [P, B] \xrightarrow{M(\alpha)} (\nu c_1, c_2) [P_2, B''] = [(\nu c) P_2, B''(\nu c/c)] \]
The last \( = \) is because \( c \notin \text{gn}(B''(\nu c/c)) \) and \( c \notin \text{ln}(B'') \).

\[ \text{Open* } P = (\nu c) P''; B' = B(c/\nu c); P'', B' \xrightarrow{\pi(c)} P', B' \] As in \text{New*},
\[ [P, B] = (\nu c_1, c_2) [P'', B'] \]
By induction,

\[
\begin{align*}
[P'', B'] & \xrightarrow{M(\overline{\alpha}(c))} [P', B'] \\
[P, B] & \xrightarrow{M(\overline{\alpha}(\nu c))} [P', B']
\end{align*}
\]

Proof. The result follows from Lemma 8. \(\Box\)

Proof of Lemma 10

If \(c \not\in \text{dom}(B)\),

\[
[P, B] \xrightarrow{\tau_1(d_1, d_2)} [P'] \ {d_1, d_2/x_1, x_2} \ | \ [B] = [P' \ {d/x}, B]
\]

If \(c \in \text{dom}(B)\), the encoding may also perform this action. However, only the buffer process \(F_{n, L}(c_1, c_2)\) is able to perform the complementary \(\tau_1(d_1, d_2)\) action, hence if \(c\) is a buffered names, we only need to consider the following transition:

\[
[P, B] \xrightarrow{\nu c} [P'] \ {d_1, d_2/x_1, x_2} \ | \ F_{n, L}(c_1, c_2) \ | \ [B \ {c}] = [P' \ {d/x}, B, B[c \mapsto (n, l)]]
\]

where \(B(c) = (n, [d] : l)\).

For output prefix. \(P = \tau(d).P'\) and \(ln(B) = \emptyset\).

\[
[P, B] = \tau_2(d_1, d_2) \ {P'} \ | \ [B] \xrightarrow{M(\alpha)} R
\]

If \(c \not\in \text{dom}(B)\),

\[
[P, B] \xrightarrow{\nu c} [P'] \ | \ [B] = [P', B]
\]

If \(c \in \text{dom}(B)\) and \(B(c) = (n, l)\) where \(ll < n\)

\[
[P, B] \xrightarrow{\nu c} [P'] \ | \ F_{n, L;I ; (d_1, d_2)}(c_1, c_2) \ | \ [B \ {c}] = [P', B, B[c \mapsto (n, l ; [d])]]
\]

For summation. \(P = \sum_{i \in I} P_i\) and \(ln(B) = \emptyset\).

\[
[P, B] = \sum_{i \in I} [P_i] \ | \ [B] \xrightarrow{M(\alpha)} R
\]

This \(M(\alpha)\) may be an action of \(\sum_{i \in I} [P_i]\) alone, or communication between \(\sum_{i \in I} [P_i]\) and \([B]\). In any case, suppose \([P_j] \xrightarrow{\alpha'} R'\), then

\[
[P_j, B] = [P_j] \ | \ [B] \xrightarrow{M(\alpha)} R
\]
By induction, $P_1, B \xrightarrow{\alpha} P_1', B'$ and $R = [P_1', B']$. By sum

$$\sum_{i \in I} P_i, B \xrightarrow{\alpha} P_i', B'$$

For parallel composition, $P = P_1 | P_2$ and $\ln(B) = \emptyset$

$$[P, B] = [P_1] | [P_2] | [B] \xrightarrow{M(\alpha)} R$$

This $M(\alpha)$ may be induced by $[P_1]$ (or $[P_2]$) alone, or communication between $[P_1]$ and $[P_2]$. For the former, suppose

$$[P_1, B] = [P_1] | [B] \xrightarrow{M(\alpha)} R'$$

By induction

$P_1, B \xrightarrow{\alpha} P_1', B'$ and $R' = [P_1', B']$

Consider the following cases regarding $B'$: if $c \in \ln(B')$, this means $P_1$ sends a local name $c$ to $B$. Then $P_1 \equiv_B (\nu c)P_1''$, $P_1' \equiv_B (\nu c)P_1'''$ and

$$[P_1, B] \xrightarrow{M(\alpha)} (\nu c_1, c_2)([P_1'''] | [B'(c/\nu c)])$$

Suppose $c$ does not occur in $P_2$ (if it does, rename $c$ to a name not occur in $P_2$), we have

$$[P, B] \xrightarrow{M(\alpha)} (\nu c_1, c_2)([P'''] | [P_2] | [B'(c/\nu c)]) = [P'', B']$$

$$P_1 | P_2 \equiv_B (\nu c)(P_1' | P_2) (\nu c)(P_1'' | P_2), B \xrightarrow{\alpha} P'', B'$$

where $P'' = (\nu c)(P_1''' | P_2)$. If $\ln(B') = \emptyset$, then

$$[P_1, B] \xrightarrow{M(\alpha)} [P_1'] | [B']$$

$$[P, B] \xrightarrow{M(\alpha)} [P_1'] | [P_2] | [B'] = [P_1' | P_2, B']$$

$$P_1 | P_2, B \xrightarrow{\alpha} P_1' | P_2, B'$$

If $M(\alpha)$ is a communication action between $[P_1]$ and $[P_2]$. Suppose

$$[P_1, B] \xrightarrow{a_{(c_1, c_2)}}, R_1 \quad [P_2, B] \xrightarrow{\tau(c_1, c_2)} R_2$$

By induction

$$P_1, B \xrightarrow{a(\alpha)} P_1', B \quad P_2, B \xrightarrow{\tau(\alpha)} P_2', B \text{ and } R_1 = [P_1', B] \quad R_2 = [P_2', B]$$

Therefore

$$P_1 | P_2, B \xrightarrow{\tau} P_1' | P_2, B$$

$$[P_1 | P_2, B] \xrightarrow{\tau} [P_1'] | [P_2'] | [B] = [P_1' | P_2', B]$$
For new process. \( P = (\nu a)P' \), Suppose \( a \notin gn(B) \)
\[
[P, B] \equiv (\nu a_1, a_2)(([P', B{a/\nu a}])] \xrightarrow{M(\alpha)} R
\]
If \( a \notin n(\alpha) \), we have
\[
[P', B{a/\nu a}] \xrightarrow{M(\alpha)} R'
\]
By induction, \( P', B{a/\nu a} \xrightarrow{\alpha} P'', B' \) and \( R' = [P'', B'] \). Since \( R = (\nu a_1, a_2)R' = (\nu a_1, a_2)[P'', B'] \). Suppose \( a \notin ln(B') \), by rule NEW*
\[
(\nu a)P', B \xrightarrow{\alpha} (\nu a)P'', B'{\nu a/a}
\]
\[
R = [(\nu a)P'', B'{\nu a/a}]
\]
Suppose \( a \in ln(B') \), this means \( P' \) outputs a new name \( \nu a \) (not the outermost \( \nu a \) of \( P \)) to the buffer, then \( a \notin ln(B) \) since \( a \notin gn(B{a/\nu a}) \), also \( a \) is not a fresh name of \( P' \). Choose a fresh name \( a' \) such that \( N(a') = (a_1', a_2') \) and \( a_1', a_2' \) are fresh names in \( [P, B] \), then \( R \equiv (\nu a_1', a_2')[P'', B'] \)
\[
(\nu a)P' \equiv_B (\nu a')P'
\]
\[
(\nu a')P', B \xrightarrow{\alpha} (\nu a')P'', B'
\]
\[
R = [(\nu a')P'', B']
\]
If \( a \in n(\alpha) \) i.e. \( \alpha = \overline{\nu a} \), we have
\[
[P', B{a/\nu a}] \xrightarrow{M(\overline{\nu a})} R
\]
By induction, \( R = [P'', B{a/\nu a}] \) and \( P', B{a/\nu a} \xrightarrow{\overline{\nu a}} P'', B{a/\nu a} \). By rule OPEN*
\[
(\nu a)P', B \xrightarrow{\alpha} P'', B{a/\nu a}
\]
Suppose \( a \in gn(B) \) (\( a \notin ln(B) \)), the above equations are also valid after appropriate \( \alpha \)-conversion of \((\nu a_1, a_2)\), and this \( \alpha \)-conversion corresponds to renaming of \( a \) in \( P \).
\[
(\nu a)P' \equiv_B (\nu a')P'{a'/a}
\]
where \( a' \) is a fresh name.
For extended new process. \( P = (\nu b : n)P' \).
\[
[P, B] = (\nu d_1 \overline{d_2})((\nu b_1, b_2)\tau(.([Q'] | F_{n,L}(b_1, b_2)) | [B{d/\nu d}])] \xrightarrow{M(\alpha)}
\]
where \( l = \lfloor . \rfloor \). The process the following transition
\[
[P, B] \xrightarrow{\tau} (\nu \overline{d_1} \overline{d_2})((\nu b_1, b_2)([Q'] | F_{n,L}(b_1, b_2)) | [B{d/\nu d}])
\]
Suppose \( b \notin \text{dom}(B) \), then \( b \notin n(B) \),
\[
P, B \xrightarrow{\tau} P', B[b \mapsto (n, l)]
\]
\[\llbracket P, B \rrbracket \xrightarrow{\tau} (\nu d_1 \tilde{d}_2)(\nu b_1, b_2)(\llbracket Q' \rrbracket | F_{n, L_1}(b_1, b_2) | \llbracket B\{d/\nu \tilde{d}\} \rrbracket) = \llbracket P', B[b \mapsto (n, l)] \rrbracket\]

Suppose \(b \in \text{dom}(B)\), the equations are also valid after appropriate \(\alpha\)-conversion of \((\nu b_1 b_2)\), and this \(\alpha\)-conversion corresponds to renaming of \(b\) in \(P\).

For replication. \(P = \!\! P' \equiv \!\! P' \mid \!\! P'\) and \(\ln(B) = \emptyset\).

\[\llbracket \!\! P', B \rrbracket = \llbracket \!\! P' \rrbracket | \llbracket B \rrbracket \equiv_B \llbracket \!\! P' \rrbracket | \llbracket \!\! P' \rrbracket | \llbracket \!\! P, B \rrbracket = \llbracket \!\! P' \mid \!\! P, B \rrbracket\]

For any process \(P\), if \(c \in \text{dom}(B)\) and \(c \notin \ln(P)\), then

\[\llbracket P, B \rrbracket = (\nu d_1 \tilde{d}_2)(\llbracket Q \rrbracket | \llbracket B\{d/\nu \tilde{d}\} \setminus c \rrbracket | F_{n, L_1}(c_1, c_2)) \xrightarrow{M(\alpha)}\]

Suppose \(B(c) = (n, l)\) and \(|l| < n\), then

\[\llbracket P, B \rrbracket \xrightarrow{c_1(d_1, d_2)} (\nu d_1 \tilde{d}_2)(\llbracket Q \rrbracket | \llbracket B\{d/\nu \tilde{d}\} \setminus c \rrbracket | F_{n, L_1:\llbracket (d_1, d_2) \rrbracket}(c_1, c_2))\]

\[P, B \xrightarrow{c(d)} P, B[c \mapsto (n, l :: [d])]\]

Suppose \(B(c) = (n, [d] :: l')\), then

\[\llbracket P, B \rrbracket \xrightarrow{\pi(d_1, d_2)} (\nu d_1 \tilde{d}_2)(\llbracket Q \rrbracket | \llbracket B\{d/\nu \tilde{d}\} \setminus c \rrbracket | F_{n, L_1'}(c_1, c_2))\]

\[P, B \xrightarrow{\pi(d)} P, B[c \mapsto (n, l')]\]

This completes the proof. \(\square\)