Language Primitives and Type Discipline for Structured Communication-Based Programming

KOHEI HONDA*, VASCO T. VASCONCELOS†, AND MAKOTO KUBO‡

ABSTRACT. We introduce basic language constructs and a type discipline as a foundation of structured communication-based concurrent programming. The constructs, which are easily translatable into the summation-less asynchronous π-calculus, allow programmers to organise programs as a combination of multiple flows of (possibly unbounded) reciprocal interactions in a simple and elegant way, subsuming the preceding communication primitives such as method invocation and rendez-vous. The resulting syntactic structure is exploited by a type discipline à la ML, which offers a high-level type abstraction of interactive behaviours of programs as well as guaranteeing the compatibility of interaction patterns between processes in a well-typed program. After presenting the formal semantics, the use of language constructs is illustrated through examples, and the basic syntactic results of the type discipline are established. Implementation concerns are also addressed.

1. Introduction

Recently, significance of programming practice based on communication among processes is rapidly increasing by the development of networked computing. From network protocols over the Internet to server-client systems in local area networks to distributed applications in the world wide web to interaction between mobile robots to a global banking system, the execution of complex, reciprocal communication among multiple processes becomes an important element in the achievement of the goals of applications. Many programming languages and formalisms have been proposed so far for the description of software based on communication. As programming languages, we have CSP [7], Ada [28], languages based on Actors [2], POOL [3], ABCL [33], Concurrent Smalltalk [32], or more recently Pict and other π-calculus-based languages [6, 23, 29]. As formalisms, we have CCS [15], Theoretical CSP [8], π-calculus [18], and other process algebras. In another vein, we have functional programming languages augmented with communication primitives, such as CML [26], dML [21], and Concurrent Haskell [12]. In these languages and formalisms, various communication primitives have been offered (such as synchronous/asynchronous message passing, remote procedure call, method-call and rendez-vous), and the description of communication behaviour is done by combining these primitives. What we observe in these primitives is that, while they do express one-time interaction between processes, there is no construct to structure a series of reciprocal interactions between two parties as such. That is, the only way to represent a series of communications following a certain scenario (think of interactions between a file server and its client) is to describe them as a collection of distinct, unrelated interactions. In applications based on complex interactions among concurrent processes, which are appearing more and more in these days, the lack of structuring methods would result in low readability and careless bugs in final programs, apart from the case when the whole communication behaviour can be simply described as, say, a one-time remote procedure call. The situation may be illustrated in comparison with the design history

*Dept. of Computer Science, University of Edinburgh, UK. †Dept. of Computer Science, University of Lisbon, Portugal. ‡Dept. of Computer Science, Chiba University of Commerce, Japan.

of imperative programming languages. In early imperative languages, programs were constructed as a bare sequence of primitives which correspond to machine operations, such as assignment and goto (consider early Fortran). As is well-known, as more programs in large size were constructed, it had become clear that such a method leads to programs without lucidity, readability or verifiability, so that the notion of structured programming was proposed in 1970's. In present days, having the language constructs for structured programming is a norm in imperative languages.

Such a comparison raises the question as to whether we can have a similar structuring method in the context of communication-based concurrent programming. Its objective is to offer a basic means to describe complex interaction behaviours with clarity and discipline at the high-level of abstraction, together with a formal basis for verification. Its key elements would, above all, consist of (1) the basic communication primitives (corresponding to assignment and arithmetic operations in the imperative setting), and (2) the structuring constructs to combine them (corresponding to "if-then-else" and "while"). Verification methodologies on their basis should then be developed.

The present paper proposes structuring primitives and the accompanying type discipline, as a basic structuring method for communication-based concurrent programming. The proposed constructs have a simple operational semantics, and various communication patterns, including those of the conventional primitives as well as those which go beyond them, are representable as their combination with clarity and rigor at the high-level of abstraction. The type discipline plays a fundamental role, guaranteeing compatibility of interaction patterns among processes via a type inference algorithm in the line of ML [19]. Concretely our proposal consists of the following key ideas.

- A basic structuring concept for communication-based programming, called *session*. A session is a chain of dyadic interactions whose collection constitutes a program. A session is designated by a private port called *channel*, through which interactions belonging to that session are performed. Channels form a distinct syntactic domain; its differentiation from the usual port names is a basic design decision we take to make the logical structure of programs explicit. Other than session, the standard structuring constructs for concurrent programming, *parallel composition*, *name hiding*, *conditional* and *recursion* are provided. In particular the combination of recursion and session allows the expression of unbounded thread of interactions as a single abstraction unit.
- Three basic communication primitives, *value passing*, *label branching*, and *delegation*. The first is the standard synchronous message passing as found in e.g. CSP or π-calculus. The second is a purified form of method invocation, deprived of value passing. The third is for passing a channel to another process, thus allowing a programmer to dynamically distribute a single session among multiple processes in a well-structured way. Together with the session structure, the combination of these primitives allows the flexible description of complex communication structures with clarity and discipline.
- A *basic type discipline* for the communication primitives, as an indispensable element of the structuring method. The typability of a program ensures two possibly communicating processes always own compatible communication patterns. For example, a procedural call has a pattern of output-input from the caller's viewpoint; then the callee should have a communication pattern of input-output. Because incompatibility of interaction patterns between processes would be one of the main reasons for bugs in communication-based programming, we believe such a type discipline has important pragmatic significance. The derived type gives high-level abstraction of interactive behaviours of a program.

Because communication between processes over the network can be done between modules written in different programming languages, the proposed communication constructs may as well be used by embedding them in various programming languages; however for simplicity we present them as a self-contained small programming language, which has been stripped to the barest minimum necessary for explanation of its novel features. Using the language, the basic concept of the proposed constructs is illustrated through programming examples. They show how extant communication primitives such as remote procedural-call and method invocation can be concisely expressed as sessions of specific patterns (hence with specific type abstractions). They also show how sessions can represent those communication structures which do not conform to the preceding primitives but which would naturally arise in practice. This suggests that the session-based program organisation may constitute a synthesis of a number of familiar as well as new programming ideas concerning communication. We also show the proposed constructs can be simply translatable into the asynchronous polyadic π-calculus with branching [31]. This suggests the feasibility of implementation in distributed environment. Yet much remains to be studied concerning the proposed structuring method, including the efficient implementation of the constructs and the accompanying reasoning principles. See Section 6 for more discussions.

The technical content of the present paper is developed on the basis of the preceding proposal [27] due to Kaku Takeuchi and the present authors. The main contributions of the present proposal in comparison with [27] are: the generalisation of session structure by delegation and recursive sessions, which definitely enlarges the applicability of the proposed structuring method; the typing system incorporating these novel concepts; representation of conventional communication primitives by the structuring constructs; and basic programming examples which show how the constructs, in particular through the use of the above mentioned novel features, can lucidly represent complex communication structures which would not be amenable to conventional communication primitives.

In the rest of the paper, Section 2 introduces the language primitives and their operational semantics. Section 3 illustrates how the primitives allow clean representation of extant communication primitives. Section 4 shows how the primitives can represent those interaction structures beyond those of the conventional communication primitives through key programming examples. Section 5 presents the typing system and establishes the basic syntactic results. Section 6 concludes with discussions on the implementation concerns, related works and further issues.

2. Syntax and Operational Semantics

2.1. Basic Concepts. The central idea in the present structuring method is a *session*. A session is a series of reciprocal interactions between two parties, possibly with branching and recursion, and serves as a unit of abstraction for describing interaction. Communications belonging to a session are done via a port specific to that session, called a *channel*. A fresh channel is generated when initiating each session, for the use in communications in the session. We use the following syntax for the initiation of a session.

$$\texttt{request}\ a(k)\ \texttt{in}\ P \quad \texttt{accept}\ a(k)\ \texttt{in}\ P \qquad \text{initiation of session}$$

The `request` first requests, via a name a, the initiation of a session as well as the generation of a fresh channel, then P would use the channel for later communications. The `accept`, on the other hand, receives the request for the initiation of a session via a, generates a new channel k, which would be used for communications in P. In the above grammar, the parenthesis (k) and the key word `in` shows the binding and its scope.

Thus, in request $a(k)$ in P, the part (k) binds the free occurrences of k in P. This convention is used uniformly throughout the present paper.

Via a channel of a session, three kinds of atomic interactions are performed: *value sending* (including name passing), *branching* and *channel passing* (or *delegation*).

$k![e_1 \cdots e_n]; P$	$k?(x_1 \cdots x_n)$ in P	data sending/receiving
$k \lhd l; P$	$k \rhd \{l_1 : P_1 \| \cdots \| l_n : P_n\}$	label selection/branching
throw $k[k']; P$	catch $k(k')$ in P	channel sending/receiving (delegation)

Data sending/receiving is the standard synchronous message passing. Here e_i denotes an expression such as arithmetic/boolean formulae as well as names. We assume variables $x_1..x_n$ are all distinct. We do not consider program passing for simplicity, cf. [11]. The *branching/selection* is the minimisation of method invocation in object-based programming. $l_1, .., l_n$ are *labels* which are assumed to be pairwise distinct. The *channel sending/receiving*, which we often call *delegation*, passes a channel which is being used in a session to another process, thus radically changing the structure of a session. Delegation is the generalisation of the concept with the same name originally conceived in the concurrent object-oriented programming [34]. See Section 4.3 for detailed discussions. In passing we note that its treatment distinct from the usual value passing is essential for both disciplined programming and a tractable type inference.

Communication primitives, organised by sessions, are further combined by the following standard constructs in concurrent programming.

$P_1 \mid P_2$	concurrent composition
$(\nu a)P$ $(\nu k)P$	name/channel hiding
if e then P else Q	conditional
def $X_1(\tilde{x}_1\tilde{k}_1) = P_1$ and $\cdots$ and $X_n(\tilde{x}_n\tilde{k}_n) = P_n$ in P	recursion

We do not need sequencing since each communication primitive already accompanies one. We also use inact, the *inaction*, which denotes the lack of action (acting as the unit of "|"). *Hiding* declares a name/channel to be local in its scope (here P). Channel hiding may not be used for usual programming, but is needed for the operational semantics presented later. In conditional, e should be a boolean expression. In recursion, X, a process variable, would occur in $P_1 ... P_n$ and P zero or more times. Identifiers in $\tilde{x}_i\tilde{k}_i$ should be pairwise distinct. We can use replication (or a single recursion) to achieve the same effect, but multiple recursion is preferable for well-structured programs.

This finishes the introduction of all language constructs we shall use in this paper. We give a simple example of a program.

$$\texttt{accept}\ a(k)\ \texttt{in}\ k![1]; k?(y)\ \texttt{in}\ P \quad | \quad \texttt{request}\ a(k)\ \texttt{in}\ k?(x)\ \texttt{in}\ k![x+1]; \texttt{inact}.$$

The first process receives a request for a new session via a, generates k, sends 1 and receives a return value via k, while the second requests the initiation of a session via a, receives the value via the generated channel, then returns the result of adding 1 to the value. Observe the compatibility of communication patterns between two processes.

2.2. **Syntax Summary.** We summarise the syntax we have introduced so far. Base sets are: *names*, ranged over by $a, b, \ldots$; *channels*, ranged over by k, k'; *variables*, ranged over by $x, y, \ldots$; *constants* (including names, integers and booleans), ranged over by $c, c', \ldots$; *expressions* (including constants), ranged over by $e, e', \ldots$; *labels*, ranged over by $l, l', \ldots$; and *process variables*, ranged over by $X, Y, \ldots$. $u, u', \ldots$ denote names and channels. Then *processes*, ranged over by $P, Q \ldots$, are given by the following grammar.

P	::=	$\texttt{request}\ a(k)\ \texttt{in}\ P$	session request
	$\mid$	$\texttt{accept}\ a(k)\ \texttt{in}\ P$	session acceptance
	$\mid$	$k![\tilde{e}];P$	data sending
	$\mid$	$k?(\tilde{x})\ \texttt{in}\ P$	data reception
	$\mid$	$k \lhd l;P$	label selection
	$\mid$	$k \rhd \{l_1 : P_1 [\!] \cdots [\!] l_n : P_n\}$	label branching
	$\mid$	$\texttt{throw}\ k[k'];P$	channel sending
	$\mid$	$\texttt{catch}\ k(k')\ \texttt{in}\ P$	channel reception
	$\mid$	$\texttt{if}\ e\ \texttt{then}\ P\ \texttt{else}\ Q$	conditional branch
	$\mid$	$P \mid Q$	parallel composition
	$\mid$	$\texttt{inact}$	inaction
	$\mid$	$(\nu u)P$	name/channel hiding
	$\mid$	$\texttt{def}\ D\ \texttt{in}\ P$	recursion
	$\mid$	$X[\tilde{e}\tilde{k}]$	process variables
D	::=	$X_1(\tilde{x}_1\tilde{k}_1) = P_1\ \texttt{and} \cdots \texttt{and}\ X_n(\tilde{x}_n\tilde{k}_n) = P_n$	declaration for recursion

The association of "|" is the weakest, others being the same. Parenthesis $(\cdot)$ denotes binders which bind the corresponding free occurrences. The standard *simultaneous substitution* is used, writing e.g. $P[\tilde{c}/\tilde{x}]$. The sets of free names/channels/variables/process variables of say P, defined in the standard way, are respectively denoted by $\mathrm{fn}(P), \mathrm{fc}(P), \mathrm{fv}(P)$ and $\mathrm{fpv}(P)$. The alpha-equality is written $\equiv_\alpha$. We also set $\mathrm{fu}(P) \stackrel{\mathrm{def}}{=} \mathrm{fc}(P) \cup \mathrm{fn}(P)$. Processes without free variables or free channels are called *programs*.

2.3. Operational Semantics. For the concise definition of the operational semantics of the language constructs, we introduce the *structural equality* $\equiv$ (cf. [4, 16]), which is the smallest congruence relation on processes including the following equations.

1. $P \equiv Q$ if $P \equiv_\alpha Q$.
2. $P \mid \texttt{inact} \equiv P, \quad P \mid Q \equiv Q \mid P, \quad (P \mid Q) \mid R \equiv P \mid (Q \mid R)$.
3. $(\nu u)\texttt{inact} \equiv \texttt{inact}, \quad (\nu uu)P \equiv (\nu u)P, \quad (\nu uu')P \equiv (\nu u'u)P, \quad (\nu u)P \mid Q \equiv (\nu u)(P \mid Q)$ if $u \notin \mathrm{fu}(Q)$, $\quad (\nu u)\texttt{def}\ D\ \texttt{in}\ P \equiv \texttt{def}\ D\ \texttt{in}\ (\nu u)P$ if $u \notin \mathrm{fu}(D)$.
4. $(\texttt{def}\ D\ \texttt{in}\ P) \mid Q \equiv \texttt{def}\ D\ \texttt{in}\ (P \mid Q)$ if $\mathrm{fpv}(D) \cap \mathrm{fpv}(Q) = \emptyset$.
5. $\texttt{def}\ D\ \texttt{in}\ (\texttt{def}\ D'\ \texttt{in}\ P) \equiv \texttt{def}\ D\ \texttt{and}\ D'\ \texttt{in}\ P$ if $\mathrm{fpv}(D) \cap \mathrm{fpv}(D') = \emptyset$.

Now the operational semantics is given by the *reduction relation* $\rightarrow$, denoted $P \rightarrow Q$, which is the smallest relation on processes generated by the following rules.

[LINK]	$(\texttt{accept}\ a(k)\ \texttt{in}\ P_1) \mid (\texttt{request}\ a(k)\ \texttt{in}\ P_2) \rightarrow (\nu k)(P_1 \mid P_2)$
[COM]	$(k![\tilde{e}];P_1) \mid (k?(\tilde{x})\ \texttt{in}\ P_2) \rightarrow P_1 \mid P_2[\tilde{c}/\tilde{x}] \qquad (\tilde{e} \downarrow \tilde{c})$
[LABEL]	$(k \lhd l_i;P) \mid (k \rhd \{l_1 : P_1 [\!] \cdots [\!] l_n : P_n\}) \rightarrow P \mid P_i \quad (1 \le i \le n)$
[PASS]	$(\texttt{throw}\ k[k'];P_1) \mid (\texttt{catch}\ k(k')\ \texttt{in}\ P_2) \rightarrow P_1 \mid P_2$
[IF1]	$\texttt{if}\ e\ \texttt{then}\ P_1\ \texttt{else}\ P_2 \rightarrow P_1 \qquad (e \downarrow \texttt{true})$
[IF2]	$\texttt{if}\ e\ \texttt{then}\ P_1\ \texttt{else}\ P_2 \rightarrow P_2 \qquad (e \downarrow \texttt{false})$
[DEF]	$\texttt{def}\ D\ \texttt{in}\ (X[\tilde{e}\tilde{k}] \mid Q) \rightarrow \texttt{def}\ D\ \texttt{in}\ (P[\tilde{c}/\tilde{x}] \mid Q)\ (\tilde{e} \downarrow \tilde{c}, X(\tilde{x}\tilde{k}) = P \in D)$
[SCOP]	$P \rightarrow P' \ \Rightarrow\ (\nu u)P \rightarrow (\nu u)P'$
[PAR]	$P \rightarrow P' \ \Rightarrow\ P \mid Q \rightarrow P' \mid Q$
[STR]	$P \equiv P'$ and $P' \rightarrow Q'$ and $Q' \equiv Q \ \Rightarrow\ P \rightarrow Q$

Above we assume the standard *evaluation relation* $\downarrow$ on expressions is given. We write $\rightarrow^*$ for the reflexive, transitive closure of $\rightarrow$. Note how a fresh channel is generated in [LINK]

rule as the result of interaction (in this way request $a(k)$ in P corresponds to the *bound output* in π-calculus[18]). In the [LABEL] rule, one of the branches is selected, discarding the remaining ones. Note we do not allow reduction under various communication prefixes, as in standard process calculi. As an example, the simple program in 2.1 has the following reduction (below and henceforth we omit trailing inactions).

$$\begin{aligned}\texttt{accept}\ a(k)\ \texttt{in}\ k![1]; k?(y)\ \texttt{in}\ P \mid \texttt{request}\ a(k)\ \texttt{in}\ k?(x)\ \texttt{in}\ k![x+1] \\ \rightarrow (\nu k)(k![1]; k?(y)\ \texttt{in}\ P \mid k?(x)\ \texttt{in}\ k![x+1]) \\ \rightarrow (\nu k)(k?(y)\ \texttt{in}\ P \mid k![x+1]) \quad \rightarrow \quad P[2/y].\end{aligned}$$

Observe how interaction proceeds in a lock-step fashion. This is due to the synchronous form of the present communication primitives.

3. Representing Communication (1) Extant Communication Primitives

This section discusses how structuring primitives can represent with ease the communication patterns of conventional communication primitives which have been in use in programming languages. They show how we can understand the extant communication patterns as a fixed way of combining the proposed primitives.

3.1. **Call-return.** The *call-return* is a widely used communication pattern in which a process calls another process, then the callee would, after some processing, returns some value to the caller. Usually the caller just waits until the reply message arrives. This concept is widely used as a basic primitive in distributed computing under the name of *remote procedure call.* We may use the following pair of notations for call-return.

$$x = \texttt{call}\ f[e_1 \cdots e_n]\ \texttt{in}\ P, \qquad \texttt{fun}\ f(x_1 \cdots x_n)\ \texttt{in}\ P.$$

On the right, the return command return[e] would occur in P. Assume these constructs are added to the syntax in 2.2. A simple programming example follows.

Example 3.1 (Factorial).

$$\begin{aligned}\mathrm{Fact}(f) = \texttt{fun}\ f(x)\ \texttt{in} \\ \texttt{if}\ x = 1\ \texttt{then return}[1] \\ \texttt{else}\ (\nu b)(\mathrm{Fact}[b] \mid y = \texttt{call}\ b[x-1]\ \texttt{in return}[x * y])\end{aligned}$$

Here and henceforth we write $X(\tilde{x}\tilde{k}) = P$, or a sequence of such equations, for the declaration part of recursion, leaving the body part implicit. This example implements the standard recursive algorithm for the factorial function. $y = \texttt{call}\ f[5]\ \texttt{in}\ P$ would give its client process.

The communication patterns based on call-return are easily representable by the combination of request/accept and send/receive. We first show the mapping of the caller. Below $[\![\cdot]\!]$ denotes the inductive mapping into the structuring primitives.

$$[\![x = \texttt{call}\ a[\tilde{e}]\ \texttt{in}\ P]\!] \stackrel{\text{def}}{=} \texttt{request}\ a(k)\ \texttt{in}\ k![\tilde{e}]; k?(x)\ \texttt{in}\ [\![P]\!].$$

Naturally we assume k is chosen fresh. The basic scenario is that a caller first initiates the session, sends the values, and waits until it receives the answer on the same channel. On the other hand, the callee is translated as follows:

$$[\![\texttt{fun}\ a(\tilde{x})\ \texttt{in}\ P]\!] \stackrel{\text{def}}{=} \texttt{accept}\ a(k)\ \texttt{in}\ k?(\tilde{x})\ \texttt{in}\ [\![P]\!][k![e]/\texttt{return}[e]].$$

Here $[k![e]/\texttt{return}[e]]$ denotes the syntactic substitution of $k![e]$ for each return[e]. Observe that the original "return" operation is expressed by the "send" primitive of

the structuring primitives. As one example, let us see how the factorial program in Example 3.1 is translated by the above mapping.

Example 3.2 (Factorial, translation).

$$\begin{aligned}\mathrm{Fact}(f) \;=\; & \texttt{accept } f(k) \texttt{ in } k?(x) \texttt{ in} \\ & \quad \texttt{if } x = 1 \texttt{ then } k![1]; \\ & \quad \texttt{else } (\nu b)(\mathrm{Fact}[b] \mid \texttt{request } b(k') \texttt{ in } k'![x-1]; k'?(y) \texttt{ in } k![x * y])\end{aligned}$$

Notice how the usage of k' and k differentiates the two contexts of communication. If we compose the translation of factorial with that of its user, we have the reduction:

$$\mathrm{Fact}[f] \mid [\![y = \texttt{call } f[3] \texttt{ in } P]\!] \;\rightarrow^*\; \mathrm{Fact}[f] \mid [\![P]\!][6/y].$$

In this way, the semantics of the synchronous call-return is given by that of the structuring primitives via the translation. Some observations follow.

(1) The significance of the specific notations for call-return would lie in the declaration of the assumed fixed communication pattern, which would enhance readability and help verification. At the same time, the translation retains the same level of clarity as the original code, even if it does need a few additional key strokes.

(2) In translation, the caller and the callee in general own complementary communication patterns, i.e. input-output meets output-input. However if, for example, the return commands appear twice in the callee, the complementarity is lost. This relates to the notion of types we discuss later: indeed, non-complementary patterns are rejected by the typing system.

(3) The translation also suggests how structuring primitives would generalise the traditional call-return structure. That is, in addition to the fixed pattern of input-output (or, correspondingly, output-input), we can have a sequence of interactions of indefinite length. For such programming examples, see Section 4.

3.2. **Method Invocation.** The idea of method invocation originates in object-oriented languages, where a caller calls an object by specifying a method name, while the object waits with a few methods together with the associated codes, so that, when invoked, executes the code corresponding to the method name. The call may or may not result in returning an answer. As a notation, an "object" would be written:

$$\texttt{obj } a{:}\{l_1(\tilde{x}_1) \texttt{ in } P_1 [\!] \cdots [\!] l_n(\tilde{x}_n) \texttt{ in } P_n\},$$

where a gives an object identifier, $l_1, \ldots, l_n$ are labels (all pairwise distinct) with formal parameters $\tilde{x}_i$, and P_i gives the code corresponding to each l_i. The return action, written $\texttt{return}[e]$, may occur in each P_i as necessary. A caller then becomes:

$$x = a.l_i[\tilde{e}] \texttt{ in } P \qquad a.l_i[\tilde{e}]; P.$$

The left-hand side denotes a process which invokes the object with a method l_i together with arguments $\tilde{e}$, then, receiving the answer, assigns the result to x, and finally executes the continuation P. The right-hand side is a notation for the case when the invocation does not need the return value. As an example, let us program a simple cell.

Example 3.3 (Cell).

$$\mathrm{Cell}(a, c) \;=\; \texttt{obj } c : \{\texttt{read}() \texttt{ in } (\texttt{return}[c] \mid \mathrm{Cell}[a, c]) [\!] \texttt{write}(x) \texttt{ in } \mathrm{Cell}[a, x]\,\}.$$

The cell $\mathrm{Cell}[a, c]$ denotes an object which saves its value in c and has a name a. There are two methods, `read` and `write`, each with the obvious functionalities. A caller which "reads" this object would be written, for example, $x = a.\texttt{read}[\,] \texttt{ in } P$.

The method invocation can be represented in the structuring constructs by combining label branching and value passing. We show the translations of an object, a caller which expects the return, and a caller which does not expect the return, in this order.

$$[\![\texttt{obj}\ a{:}\{l_1(\tilde{x}_1)\ \texttt{in}\ P_1 \| \cdots \| l_n(\tilde{x}_n)\ \texttt{in}\ P_n\}]\!] \stackrel{\text{def}}{=}$$
$$\texttt{accept}\ a(k)\ \texttt{in}\ k \rhd \{l_1 : k?(\tilde{x}_1)\ \texttt{in}\ [\![P_1]\!]\sigma \| \cdots \| l_n : k?(\tilde{x}_n)\ \texttt{in}\ [\![P_n]\!]\sigma\}$$
$$[\![x = a.l_i[\tilde{e}]\ \texttt{in}\ P]\!] \stackrel{\text{def}}{=} \texttt{request}\ a(k)\ \texttt{in}\ k \lhd l_i; k![\tilde{e}]; k?(x)\ \texttt{in}\ [\![P]\!]$$
$$[\![a.l_i[\tilde{e}]; P]\!] \stackrel{\text{def}}{=} \texttt{request}\ a(k)\ \texttt{in}\ k \lhd l_i; k![\tilde{e}]; [\![P]\!]$$

In each translation, k should be fresh. In the first equation, σ denotes the substitution $[k![e]/\texttt{return}[e]]$, replacing all occurrences of $\texttt{return}[e]$ by $k![e]$. Observe that a method invocation is decomposed into label branching and value passing. Using this mapping, the cell is now translated into:

Example 3.4 (Cell, translation).

$$\text{Cell}(a,c) = \texttt{accept}\ a(k)\ \texttt{in}\ k \rhd \{\texttt{read} : k![e]; \text{Cell}[a,c] \parallel \texttt{write} : k?(x)\ \texttt{in}\ \text{Cell}[a,x]\}.$$

Similarly, $x = a.\texttt{read}[]\ \texttt{in}\ P$ is translated as $\texttt{request}\ c(k)\ \texttt{in}\ k \lhd \texttt{read}; k?(x)\ \texttt{in}\ P$, while $a.\texttt{write}[3]$ becomes $\texttt{request}\ a(k)\ \texttt{in}\ k \lhd \texttt{write}; k![3]$. Some observations follow.

(1) The translation is not much more complex than the original text. The specific notation however has a role of declaring the fixed communication pattern and saves key-strokes.

(2) Here again, the translation of the caller and the callee in general results in complementary patterns. There are however two main cases where the complementarity is lost, one due to the existence/lack of the return statement (e.g. $x = a.\texttt{write}[3]\ \texttt{in}\ Q$ above) and another due to an invocation of an object with a non-existent method. Again such inconsistency is detectable by a typing system introduced later.

(3) The translation suggests how structuring primitives can generalise the standard method invocation. For example, an object may in turn invoke the method of the caller after being invoked. We believe that, in programming practice based on the idea of interacting objects, such reciprocal, continuous interaction would arise naturally and usefully. Such examples are discussed in the next section.

In addition to call-return and method invocation, we can similarly represent other communication patterns in use with ease, for example asynchronous call-return (which includes *rendez-vous* [28] and *future* [2]) and simple message passing. For the space sake we leave their treatment to [11]. Section 6 gives a brief discussion on the significance of specific notations for these fixed communication patterns in language design.

4. Representing Communication (2) Complex Communication Patterns

This section shows how the structuring primitives can cleanly represent various complex communication patterns which go beyond those represented in conventional communication primitives.

4.1. **Continuous Interactions.** The traditional call-return primitive already encapsulates a sequence of communications, albeit simple, as a single abstraction unit. The key possibility of the session structure lies in that it extends this idea to arbitrarily complex communication patterns, including the case when multiple interaction sequences interleave with each other. The following shows one such example, which describes the behaviour of a banking service to the user (essentially an automatic teller machine).

Example 4.1 (ATM).

$$\begin{aligned}
\mathrm{ATM}(a,b) = {} & \texttt{accept}\ a(k)\ \texttt{in}\ k![id]; \\
& k \rhd \{\texttt{deposit} : \texttt{request}\ b(h)\ \texttt{in}\ k?(amt)\ \texttt{in} \\
& \quad h \lhd \texttt{deposit}; h![id, amt]; \mathrm{ATM}[a,b] \\
& \quad \|\texttt{withdraw} : \texttt{request}\ b(h)\ \texttt{in}\ k?(amt)\ \texttt{in} \\
& \quad h \lhd \texttt{withdraw}; h![id, amt]; \\
& \quad h \rhd \{\texttt{success} : k \lhd \texttt{dispense}; k![amt]; \mathrm{ATM}[a,b] \\
& \quad\quad \|\texttt{failure} : k \lhd \texttt{overdraft}; \mathrm{ATM}[a,b]\} \\
& \quad \|\texttt{balance} : \texttt{request}\ b(h)\ \texttt{in}\ h \lhd \texttt{balance}; h?(amt)\ \texttt{in} \\
& \quad k![amt]; \mathrm{ATM}[a,b]\}
\end{aligned}$$

The program, after establishing a session with the user via a, first lets the user input the user code (we omit such details as verification of the code etc.), then offers three choices, `deposit`, `withdraw`, and `balance`. When the user selects one of them, the corresponding code is executed. For example, when the `withdraw` is chosen, the program lets the user enter the amount to be withdrawn, then interacts with the bank via b, asking with the user code and the amount. If the bank answers there is enough amount in the account, the money is given to the user. If not, then the `overdraft` message results. In either case the system eventually returns to the original waiting mode. Note in particular the program should communicate with the bank in the midst of interaction with the user, so three parties are involved in interaction as a whole. A user may be written as:

$$\begin{aligned}
& \texttt{request}\ a(k)\ \texttt{in}\ k![myId]; k \lhd \texttt{withdraw}; k![58]; \\
& \qquad\qquad k \rhd \{\texttt{dispense} : k?(amt)\ \texttt{in}\ P \| \texttt{overdraft} : Q\}.
\end{aligned}$$

Here we may consider Q as a code for "exception handling" (invoked when the balance is less than expected). Notice also interactions are now truly reciprocal.

4.2. **Unbounded Interactions.** The previous example shows how structuring primitives can easily describe the situation which would be difficult to program in a clean way using the conventional primitives. At the same time, the code does have room for amelioration. If a user wants to look at his balance before he withdraws, he should enter his user code twice. The following refinement makes this redundancy unnecessary.

Example 4.2 (Kind ATM).

$$\begin{aligned}
\mathrm{ATM}'(a,b) = {} & \texttt{accept}\ a(k)\ \texttt{in}\ k![id]; \mathrm{Actions}[a,b,id,k] \\
\mathrm{Actions}(a,b,id,k) = {} & k \rhd \{\texttt{deposit} : \texttt{request}\ b(h)\ \texttt{in}\ k?(amt)\ \texttt{in} \\
& \quad h \lhd \texttt{deposit}; h![id, amt]; \mathrm{Actions}[a,b,id,k] \\
& \quad \|\texttt{withdraw} : \texttt{request}\ b(h)\ \texttt{in}\ k?(amt)\ \texttt{in} \\
& \quad h \lhd \texttt{withdraw}; h![id]; h![amt]; \\
& \quad h \rhd \{\texttt{success} : k \lhd \texttt{dispense}; k![amt]; \mathrm{Actions}[a,b,id,k] \\
& \quad\quad \|\texttt{failure} : k \lhd \texttt{overdraft}; \mathrm{Actions}[a,b,id,k]\} \\
& \quad \|\texttt{balance} : \texttt{request}\ b(h)\ \texttt{in}\ h \lhd \texttt{balance}; h?(amt)\ \texttt{in} \\
& \quad k![amt]; \mathrm{Actions}[a,b,id,k] \\
& \quad \|\texttt{quit} : \mathrm{ATM}'[a,b]\}
\end{aligned}$$

As can be seen, the main difference lies in that the process is still within the same session even after recurring to the waiting mode, after processing each request: once a user establishes the session and enters the user code, she can request various services as many times as she likes. To exit from this loop, the branch quit is introduced.

This example shows how recursion within a session allows a flexible description of interactive behaviour which goes beyond the recursion in usual objects (where each session consists of a fixed number of interactions). It is notable that the unbounded session owns a rigorous syntactic structure so as to allow type abstraction, see Section 5. Unbounded interactions with a fixed pattern naturally arise in practice, e.g. interactions between a file server and its client. We believe recursion within a session can be effectively used for varied programming practice.

4.3. **Delegation.** The original idea of delegation in object-based concurrent programming [34] allows an object to delegate the processing of a request it receives to another object. Its basic purpose is *distribution of processing*, while maintaining the transparency of name space for clients of that service. Practically it can be used to enhance modularity, to express exception handing, and to increase concurrency. The following example shows how we can generalise the original notion to the present setting.

Example 4.3 (Ftp server). Below $\bigotimes_n P$ denotes the n-fold parallel composition.

$$
\begin{aligned}
\mathrm{Init}(pid, nis) &= (\nu b)(\mathrm{Ftpd}[pid, b] \mid \textstyle\bigotimes_n \mathrm{FtpThread}[b, nis]) \\
\mathrm{Ftpd}(pid, b) &= \mathtt{accept}\ pid(s)\ \mathtt{in}\ \mathtt{request}\ b(k)\ \mathtt{in}\ \mathtt{throw}\ k[s]; \mathrm{Ftpd}[pid, b] \\
\mathrm{FtpThread}(b, nis) &= \mathtt{accept}\ b(k)\ \mathtt{in}\ \mathtt{catch}\ k(s)\ \mathtt{in}\ s?(userid, passwd)\ \mathtt{in} \\
&\qquad \mathtt{request}\ nis(j)\ \mathtt{in}\ j \lhd \mathtt{checkUser}; j![userid, passwd]; \\
&\qquad j \rhd \{\mathtt{invalid} : s \lhd \mathtt{sorry} \| \mathtt{valid} : s \lhd \mathtt{welcome}; \mathrm{Actions}[s, b]\} \\
\mathrm{Actions}(s, b) &= s \rhd \{\mathtt{get} : \cdots ; s?(file)\ \mathtt{in}\ \cdots ; \mathrm{Actions}[s, b] \\
&\qquad \| \mathtt{put} : \cdots ; s?(file)\ \mathtt{in}\ \cdots ; \mathrm{Actions}[s, b] \\
&\qquad \| \mathtt{bye} : \cdots ; \mathrm{FtpThread}[b, nis]\}.
\end{aligned}
$$

The above code shows an outline of the code of a ftp server, which follows the behaviour of the standard ftp servers with TCP/IP protocol. Initially the program Init generates a server Ftpd and n threads with the identical behaviour (for simplicity we assume all threads share the same name b). Suppose, in this situation, a server receives a request for a service from some client, establishing the session channel s. A server then requests for another session with an idle thread (if it exists) and "throws" a channel s to that thread, while getting ready for the next request from clients itself. It is now the thread FtpThread which actually processes the user's request, receiving the user name, referring to NIS, and executing various operations (note recursion within a session is used). Here the delegation is used to enable the ftp server to process multiple requests concurrently without undue delay in response. The scheme is generally applicable to a server interacting with many clients. Some observations follow.

(1) The example shows how the generalised delegation allows programmers to cleanly describe those interaction patterns which generalise the original form of delegation. Other examples of the usage of delegation abound, for example a file server with geographically distributed sites or a server with multiple services each to be processed by a different sub-server.

(2) A key factor of the above code is that a *client does not have to be conscious of the delegation which takes place on the server's side*: that is, a client program can be

written as if it is interacting with a single entity, for example as follows.

$$\texttt{request}\ pid(s)\ \texttt{in}\ s![myId]; s \triangleright \{\texttt{sorry} : \cdots \| \texttt{welcome} : \cdots\}$$

Observe that, between the initial `request` and the next sending operation, the `catch/throw` interaction takes place on the server's side: however the client process does not have to be conscious of the event. This shows how delegation enables distribution of computation while maintaining the transparency of the name space.

(3) If we allow each ftp-thread to be dynamically generated, we can use parallel composition to the same effect, just as the use of "fork" to pass process resources in UNIX. While this scheme has a limitation in that we cannot send a channel to an already running process, it offers another programming method to realise flexible, dynamic communication structures. We also observe that the use of `throw/catch`, or the "fork" mentioned above, would result in complexly woven sequences of interactions, which would become more error-prone than without. In such situations, the type discipline discussed in the next section would become an indispensable tool for programming, where we can algorithmically verify if a program has coherent communication structure and, in particular, if it contains interaction errors.

5. The Type Discipline

5.1. **Preliminaries.** The present structuring method allows the clear description of complex interaction structures beyond conventional communication primitives. The more complex the interaction becomes, however, the more difficult it would be to capture the whole interactive behaviour and to write correct programs. The type discipline we shall discuss in this section gives a simple solution to these issues at a basic level. We first introduce the basic notions concerning types, including duality on types which represents complementarity of interactions.

Definition 5.1 (Types). Given *type variables* $(t, t', \ldots)$ and *sort variables* $(s, s', \ldots)$, *sorts* $(S, S', \ldots)$ and *types* $(\alpha, \beta, \ldots)$ are defined by the following grammar.

$$\begin{aligned} S &::= \mathbf{nat} \mid \mathbf{bool} \mid \langle \alpha, \overline{\alpha} \rangle \mid s \mid \mu s.S \\ \alpha &::= \downarrow[\tilde{S}]; \alpha \mid \downarrow[\alpha]; \beta \mid \&\{l_1 : \alpha_1, \ldots, l_n : \alpha_n\} \mid \mathbf{1} \mid \bot \\ &\quad \mid \uparrow[\tilde{S}]; \alpha \mid \uparrow[\alpha]; \beta \mid \oplus\{l_1 : \alpha_1, \ldots, l_n : \alpha_n\} \mid t \mid \mu t.\alpha \end{aligned}$$

where, for a type α in which $\bot$ does not occur, we define $\overline{\alpha}$, the *co-type* of α, by:

$$\overline{\uparrow[\tilde{S}]; \alpha} = \downarrow[\tilde{S}]; \overline{\alpha} \quad \overline{\oplus\{l_i : \alpha_i\}} = \&\{l_i : \overline{\alpha_i}\} \quad \overline{\uparrow[\alpha]; \beta} = \downarrow[\alpha]; \overline{\beta} \quad \overline{\mathbf{1}} = \mathbf{1}$$

$$\overline{\downarrow[\tilde{S}]; \alpha} = \uparrow[\tilde{S}]; \overline{\alpha} \quad \overline{\&\{l_i : \alpha_i\}} = \oplus\{l_i : \overline{\alpha_i}\} \quad \overline{\downarrow[\alpha]; \beta} = \uparrow[\alpha]; \overline{\beta} \quad \overline{t} = t \quad \overline{\mu t.\alpha} = \mu t.\overline{\alpha}.$$

A *sorting* (resp. a *typing*, resp. a *basis*) is a finite partial map from names and variables to sorts (resp. from channels to types, resp. from process variables to the sequences of sorts and types). We let $\Gamma, \Gamma', \ldots$ (resp. $\Delta, \Delta', \ldots$, resp. $\Theta, \Theta', \ldots$) range over sortings (resp. typings, resp. bases). We regard types and sorts as representing the corresponding regular trees in the standard way [5], and consider their equality in terms of such representation.

A sort of form $\langle \alpha, \overline{\alpha} \rangle$ represents two complementary structures of interaction which are associated with a name (one denoting the behaviour starting with `accept`, another that which starts with `request`), while a type gives the abstraction of interaction to be done through a channel. Note $\overline{\overline{\alpha}} = \alpha$ holds whenever $\overline{\alpha}$ is defined. $\bot$ is a specific type indicating no further connection is possible at a given name. The following partial algebra on the set of typings, cf. [10], plays a key role in our typing system.

Definition 5.2 (Type algebra). Typings Δ_1 and Δ_2 are *compatible*, written $\Delta_1 \asymp \Delta_2$, if $\Delta_1(k) = \overline{\Delta_2(k)}$ for all $k \in \mathrm{dom}(\Delta_1) \cap \mathrm{dom}(\Delta_2)$. When $\Delta_1 \asymp \Delta_2$, the *composition of* Δ_1 *and* Δ_2, written $\Delta_1 \circ \Delta_2$, is given as a typing such that $(\Delta_1 \circ \Delta_2)(k)$ is (1) $\bot$, when $k \in \mathrm{dom}(\Delta_1) \cap \mathrm{dom}(\Delta_2)$; (2) $\Delta_i(k)$, if $k \in \mathrm{dom}(\Delta_i) \setminus \mathrm{dom}(\Delta_{i+1 \bmod 2})$ for $i \in \{1,2\}$; and (3) undefined otherwise.

Compatibility means each common channel k is associated with complementary behaviours, thus ensuring the interaction on k to run without errors. When composed, the type for k becomes $\bot$, preventing further connection at k (note $\bot$ has no co-type). One can check the partial operation $\circ$ is partially commutative and associative.

5.2. **Typing System.** The main sequent of our typing system has a form

$$\Theta; \Gamma \vdash P \triangleright \Delta$$

which reads: "under the environment $\Theta; \Gamma$, a process P has a typing Δ." Sorting Γ specifies protocols at the free names of P, while typing Δ specifies P's behaviour at its free channels. When P is a program, Θ and Δ become both empty.

Given a typing or a sorting, say Φ, write $\Phi \cdot s : p$ for $\Phi \cup \{s : p\}$ together with the condition that $s \notin \mathrm{dom}(\Phi)$; and $\Phi \backslash s$ for the result of taking off $s : \Phi(s)$ from Φ if $\Phi(s)$ is defined (if not we take Φ itself). Also assume given the evident inference rules for arithmetic and boolean expressions, whose sequent has the form $\Gamma \vdash e \triangleright \alpha$, enjoying the standard properties such as $\Gamma \vdash e \triangleright S$ and $e \downarrow c$ imply $\Gamma \vdash c \triangleright S$. The main definition of this section follows.

Definition 5.3 (Basic typing system). The typing system is defined by the axioms and rules in Figure 1, where we assume the range of Δ in [INACT] and [VAR] contains only $\mathbf{1}$ and $\bot$.

For simplicity, the rule [DEF] is restricted to single recursion, which is easily extendible to multiple recursion. If $\Theta; \Gamma \vdash P \triangleright \Delta$ is derivable in the system, we say P *is typable under* $\Theta; \Gamma$ *with* Δ, or simply P *is typable*. Some comments on the typing system follow.

(1) In the typing system, the left-hand side of the turnstile is for shared names and variables ("classical realm"), while the right-hand side is for channels sharable only by two complementary parties (a variant of "linear realm"). It differs from various sorting disciplines in that a channel k is in general ill-sorted, e.g. it may carry an integer at one time and a boolean at another. In spite of this, the manipulation of linear realm by typing algebra ensures linearised usage of channels, as well as preventing interaction errors, cf. Theorem 5.4 below.

(2) In [THR], the behaviour represented by α for channel k' is actually performed by the process which "catches" k' (note k' cannot occur free in Δ, hence neither in P, by our convention on "$\cdot$"). To capture the interactions at k' as a whole, $k' : \alpha$ is added to the linear realm. On the other hand, the rule [CAT] guarantees that the receiving side does use the channel k' as is prescribed. Reading from the conclusions to the antecedents, [THR] and [CAT] together illustrate how k' is "thrown" from the left to the right.

(3) The simplicity of the typing rules is notable, utilising the explicit syntactic structure of session. In particular it is syntax-directed and has a principal type when we use a version of kinding [22]. It is then easy to show there is a typing algorithm à la ML, which extracts the principal type of a given process iff it is typable. It should be noted that simplicity and tractability of typing rules do *not* mean that the obtainable type information is uninteresting: the resulting type abstraction richly represents the interactive behaviour of programs, as later examples exhibit.

$$[\textsc{Acc}]\ \frac{\Theta;\Gamma \vdash P \triangleright \Delta \cdot k : \alpha}{\Theta;\Gamma, a : \langle \alpha, \overline{\alpha} \rangle \vdash \mathtt{accept}\ a(k)\ \mathtt{in}\ P \triangleright \Delta} \quad [\textsc{Req}]\ \frac{\Theta;\Gamma \vdash P \triangleright \Delta \cdot k : \overline{\alpha}}{\Theta;\Gamma, a : \langle \alpha, \overline{\alpha} \rangle \vdash \mathtt{request}\ a(k)\ \mathtt{in}\ P \triangleright \Delta}$$

$$[\textsc{Send}]\ \frac{\Gamma \vdash \tilde{e} \triangleright \tilde{S} \quad \Theta;\Gamma \vdash P \triangleright \Delta \cdot k : \alpha}{\Theta;\Gamma \vdash k![\tilde{e}]; P \triangleright \Delta \cdot k : \uparrow[\tilde{S}]; \alpha} \quad [\textsc{Rcv}]\ \frac{\Theta;\Gamma \cdot \tilde{x} : \tilde{S} \vdash P \triangleright \Delta \cdot k : \alpha}{\Theta;\Gamma \vdash k?(\tilde{x})\ \mathtt{in}\ P \triangleright \Delta \cdot k : \downarrow[\tilde{S}]; \alpha}$$

$$[\textsc{Br}]\ \frac{\Theta;\Gamma \vdash P_1 \triangleright \Delta \cdot k : \alpha_1 \quad \cdots \quad \Theta;\Gamma \vdash P_n \triangleright \Delta \cdot k : \alpha_n}{\Theta;\Gamma \vdash k \triangleright \{l_1 : P_1 [\!] \cdots [\!] l_n : P_n\} \triangleright \Delta \cdot k : \&\{l_1 : \alpha_1, \ldots, l_n : \alpha_n\}}$$

$$[\textsc{Sel}]\ \frac{\Theta;\Gamma \vdash P \triangleright \Delta \cdot k : \alpha_j}{\Theta;\Gamma \vdash k \lhd l_j; P \triangleright \Delta \cdot k : \oplus\{l_1 : \alpha_1, \ldots, l_n : \alpha_n\}} \quad (1 \le j \le n)$$

$$[\textsc{Thr}]\ \frac{\Theta;\Gamma \vdash P \triangleright \Delta \cdot k : \beta}{\Theta;\Gamma \vdash \mathtt{throw}\ k[k']; P \triangleright \Delta \cdot k : \uparrow[\alpha]; \beta \cdot k' : \alpha} \quad [\textsc{Cat}]\ \frac{\Theta;\Gamma \vdash P \triangleright \Delta \cdot k : \beta \cdot k' : \alpha}{\Theta;\Gamma \vdash \mathtt{catch}\ k(k')\ \mathtt{in}\ P \triangleright \Delta \cdot k : \downarrow[\alpha]; \beta}$$

$$[\textsc{Conc}]\ \frac{\Theta;\Gamma \vdash P \triangleright \Delta \quad \Theta;\Gamma \vdash Q \triangleright \Delta'}{\Theta;\Gamma \vdash P \mid Q \triangleright \Delta \circ \Delta'} (\Delta \asymp \Delta') \quad [\textsc{If}]\ \frac{\Gamma \vdash e \triangleright \mathbf{bool} \quad \Theta;\Gamma \vdash P \triangleright \Delta \quad \Theta;\Gamma \vdash Q \triangleright \Delta}{\Theta;\Gamma \vdash \mathtt{if}\ e\ \mathtt{then}\ P\ \mathtt{else}\ Q \triangleright \Delta}$$

$$[\textsc{NRes}]\ \frac{\Theta;\Gamma \cdot a : S \vdash P \triangleright \Delta}{\Theta;\Gamma \vdash (\nu a)P \triangleright \Delta} \quad [\textsc{CRes}]\ \frac{\Theta;\Gamma \vdash P \triangleright \Delta \cdot k : \bot}{\Theta;\Gamma \vdash (\nu k)P \triangleright \Delta} \quad [\textsc{Inact}]\ \Theta;\Gamma \vdash \mathtt{inact} \triangleright \Delta$$

$$[\textsc{Var}]\ \frac{\Gamma \vdash \tilde{e} \triangleright \tilde{S}}{\Theta, X : \tilde{S}\tilde{\alpha};\Gamma \vdash X[\tilde{e}\tilde{k}] \triangleright \Delta \cdot \tilde{k} : \tilde{\alpha}} \quad [\textsc{Def}]\ \frac{\Theta;\Gamma \cdot \tilde{x} : \tilde{S} \vdash P \triangleright \tilde{k} : \tilde{\alpha} \quad \Theta;\Gamma \vdash Q \triangleright \Delta}{\Theta \backslash X;\Gamma \vdash \mathtt{def}\ X(\tilde{x}\tilde{k}) = P\ \mathtt{in}\ Q \triangleright \Delta} (\Theta(X) = \tilde{S}\tilde{\alpha})$$

FIGURE 1. The typing system

Below we briefly summarise the fundamental syntactic properties of the typing system. We need the following notion: a *k-process* is a prefixed process with subject k (such as $k![\tilde{e}]; P$ and $\mathtt{catch}\ k(k')\ \mathtt{in}\ P$). Next, a *k-redex* is a pair of dual k-processes composed by $|$, i.e. either of forms $(k![\tilde{e}]; P \mid k?(x)\ \mathtt{in}\ Q)$, $(k \lhd l; P \mid k \triangleright \{l_1 : Q_1 [\!] \cdots [\!] l_n : Q_n\})$, or $(\mathtt{throw}\ k[k']; P \mid \mathtt{catch}\ k(k'')\ \mathtt{in}\ Q)$. Then P is an *error* if $P \equiv \mathtt{def}\ D\ \mathtt{in}\ (\nu\tilde{u})(P' | R)$ where P' is, for some k, the $|$-composition of *either* two k-processes that do not form a k-redex, *or* three or more k-processes. We then have:

Theorem 5.4.

1. (Invariance under $\equiv$) $\Theta;\Gamma \vdash P \triangleright \Delta$ *and* $P \equiv Q$ *imply* $\Theta;\Gamma \vdash Q \triangleright \Delta$.
2. (Subject reduction) $\Theta;\Gamma \vdash P \triangleright \Delta$ *and* $P \to^* Q$ *imply* $\Theta;\Gamma \vdash Q \triangleright \Delta$.
3. (Lack of run-time errors) *A typable program never reduces into an error.*

We omit the proofs, which are straightforward due to the syntax-directed nature of the typing rules. See [11] for details. We note that we can easily extend the typing system with ML-like polymorphism for recursion, which is useful for e.g. template processes (such as $\mathtt{def}\ \mathrm{Cell}(cv) = \cdots\ \mathtt{in}\ \mathrm{Cell}[a\ 42] \mid \mathrm{Cell}[b\ \mathtt{true}]$), with which all the properties in Theorem 5.4 are preserved. This and other basic extensions are discussed in [11].

5.3. **Examples.** We give a few examples of typing, taking programs in the preceding sections. We omit the final $\mathbf{1}$ from the type, e.g. we write $\downarrow[\alpha]$ for $\downarrow[\alpha]; \mathbf{1}$. First, the factorial in Example 3.2 is assigned, at its free name, a type $\downarrow[\mathbf{nat}]; \uparrow[\mathbf{nat}]$ (for factorial) and its dual $\uparrow[\mathbf{nat}]; \downarrow[\mathbf{nat}]$ (for its user). Next, the cell in Example 3.3 is given a type

$\&\{\texttt{read}:\uparrow[\alpha], \texttt{write}:\downarrow[\alpha]\}$ (for the cell) and its dual $\oplus\{\texttt{read}:\downarrow[\alpha], \texttt{write}:\uparrow[\alpha]\}$ (for its user). The type of a cell says a cell waits with two options, and, when "read" is selected, it would send an integer and the session ends, and when "write" is selected, it would receive an integer and again the session ends: its dual says a user may do either "read" or "write", and, according to which of them it selects, it behaves as prescribed.

As a more interesting example, take the "kind ATM" in Example 4.2. Consider $\mathrm{ATM}'[ab]$ under the declaration in the example. Then a typing $a : \langle \alpha, \overline{\alpha} \rangle$, $b : \langle \beta, \overline{\beta} \rangle$ is given to the process, where we set α, which abstracts the interaction with a user, as:

$$\begin{aligned}\alpha \stackrel{\text{def}}{=} \downarrow[\mathbf{nat}]; \mu t.\&\{&\texttt{deposit}:\downarrow[\mathbf{nat}]; t,\\ &\texttt{withdraw}:\downarrow[\mathbf{nat}];\ \oplus\{\texttt{dispense}:\uparrow[\mathbf{nat}]; t,\ \texttt{overdraft}: t\},\\ &\texttt{balance}:\uparrow[\mathbf{nat}]; t,\\ &\texttt{quit}:\uparrow[\mathbf{nat}]\},\end{aligned}$$

while $\overline{\beta}$, which abstracts the interaction with the banking system, is given as:

$$\begin{aligned}\overline{\beta} \stackrel{\text{def}}{=} \oplus\{&\texttt{deposit}:\uparrow[\mathbf{nat}\,\mathbf{nat}],\\ &\texttt{withdraw}:\uparrow[\mathbf{nat}\,\mathbf{nat}];\ \&\{\texttt{success}:\mathbf{1}, \texttt{failure}:\mathbf{1}\},\\ &\texttt{balance}:\uparrow[\mathbf{nat}]; \downarrow[\mathbf{nat}]\}.\end{aligned}$$

Notice the type abstraction is given separately for the user (at a) and the bank (at b), describing the behaviour of ATM' for each of its interacting parties.

As a final example, the ftp server of Example 4.3 is given the following type at its principal name:

$$\downarrow[\mathbf{nat}\,\mathbf{nat}];\ \oplus\{\texttt{sorry}:\mathbf{1},\ \texttt{welcome}:\mu t.\ \&\{\texttt{get}:\cdots; t,\ \texttt{put}:\cdots; t,\ \texttt{bye}:\cdots\}\}.$$

This example shows that the throw/catch action is abstracted away in the type with respect to the user (but not in the type with respect to the thread, which we omit) so that the user can interact without concerning himself with the delegation occurring on the other's side. In these ways, not only the type discipline offers the correctness verification of programs at a basic level, but also it gives a clean abstraction of interactive behaviours of programs, which would assist programming activities.

6. Discussions

6.1. **Implementation Concerns.** In the previous sections, we have seen how the session structure enables clean description of complex communication behaviours, employing the synchronous form of interactions as its essential element. For implementation of the primitives, however, the use of asynchronous communication is essential, since the real distributed computing environments are inherently asynchronous. To study such implementation in a formal setting, which should then be applied to the realistic implementation, we consider a translation of the present language primitives into TyCO[14], a sorted summation-less polyadic asynchronous π-calculus with branching structure (which is ultimately translatable into its monadic, branch-less version). The existence of the branching structure makes TyCO an ideal intermediate language. For the space sake we cannot discuss the technical details of the translation, for which the reader may refer to [11]. We only list essential points.

(1) The translation converts both channels and names into names. Each synchronous interaction (including the branching-selection) is translated into two asynchronous interactions, the second one acting as acknowledgement. This suggests the primitives are amenable for distributed implementation, at least at the rudimentary level.

(2) In spite of (1) above, the resulting code is far from optimal: as a simple example, if a value is sent immediately after the request operation, clearly the value can be "piggy-backed" to the request message. A related example is the translation of the factorial in Section 3 in comparison with its standard "direct" encoding in Pict or TyCO in a continuation-passing style, cf. [24]. To find the effective, well-founded optimisation methods in this line would be a fruitful subject of study (see 6.2 below).

(3) The encoding translates the typable programs into well-sorted TyCO codes. It is an intriguing question how we can capture, in a precise way, the well-typedness of the original code at the level of TyCO (this question was originally posed by Simon Gay for a slightly different kind of translation).

6.2. **Related Works and Further Issues.** In the following, comparisons with a few related works are given along with some of the significant further issues.

First, in the context of conventional concurrent programming languages, the key departure of the proposed framework lies in that the session structure allows us to form an arbitrary complex interaction pattern as a unit of abstraction, rather than being restricted to a fixed repertoire. Examples in Section 4 show how this feature results in clean description of complex communication behaviours which may not be easy to express in conventional languages. For example, if we use, for describing those examples, so-called *concurrent object-oriented languages* [34], whose programming concept based on objects and their interaction is proximate to the present one, we need to divide each series of interactions into multiple chunks of independent communications, which, together with the nesting of method invocations, would make the resulting programs hard to understand. The explicit treatment of session also enables the type-based verification of compatibility of communication patterns, which would facilitate the writing of correct communication-based programs at an elementary level.

In spite of these observations, we believe that various communication primitives in existing programming languages, such as object-invocation and RPC, would not diminish their significance even when the present primitives are incorporated. We already discussed how they would be useful for declaring fixed communication patterns, as well as for saving key strokes (in particular the primitives for simple message passing would better be given among language constructs since their description in the structuring primitives is, if easy, roundabout). Notice we can still maintain the same type discipline by regarding these constructs as standing for combination of structuring primitives, as we did in Section 3. In another vein, the communication structures which we can extract from such declaration would give useful information for performing optimisation.

Secondly, the field of research which is closely related to the present work is the study of various typed programming languages based on π-calculi [6, 23, 29], and, more broadly, the study of types for π-calculi (see for example [17, 24, 25, 13, 30, 35]). In 6.1, we already noted that the proposed primitives are easily translatable into TyCO, hence into π-calculus. Indeed, the encoding of various interaction structures in π-calculus is the starting point of the present series of study (cf. Section 2 of [27]). Comparisons with these works may be done from two distinct viewpoints.

(1) From the viewpoint of language design, Pict and other π-calculus-based languages use the primitives of (polyadic) asynchronous π-calculus or its variants as the basic language constructs, and build further layers of abstraction on their basis. The present proposal differs in that it incorporates the session-based structure as a fundamental stratum for programming, rather than relying on chains of direct name passing for describing communication behaviour. While the former is operationally translatable into the latter as discussed in 6.1, the very translation of, say,

the programming examples in the preceding sections would reveal the significance of the session structure for abstraction concerns. In particular, any such translation should use multiple names for actions belonging to a single session, which damages the clarity and readability of the program. We note that we are far from claiming that the proposed framework would form the sole stratum for high-level communication-based programming: abstraction concerns for distributed computing are so diverse that any single framework cannot meet all purposes. However we do believe that the proposed constructs (with possible variations) would offer a basic and useful building block for communication-based programming, especially when concerned communication behaviours tend to become complex.

(2) From the viewpoint of type disciplines, one notable point would be that well-typed programs in the present typing system are in general ill-sorted (in the sense of [17]) when they are regarded as π-terms, since the same channel k may be used for carrying values of different sorts at different occasions. This is in a sharp contrast with most type disciplines for π-calculus in literature. An exception is the typing system for monadic π-calculus presented in [35], where, as in the present setting, the incorporation of sequencing information in types allows certain ill-sorted processes to be well-typed. Apart from a notable difference in motivation, a main technical difference is that the rules in [35] are not syntax-directed; at the same time, the type discipline in [35] guarantees a much stronger behavioural property. Another notable feature of the present typing system is the linear usage of channels it imposes on programs. In this context, the preceding study on linearity in π-calculus such as [10, 13] offers the clarification of the deterministic character of interaction at channels (notice interaction at names is still non-deterministic in general). Also [THR] and [CAT] rules have some resemblance to the rules for linear name communication presented in [10, 13]. Regarding these and other works on types for π-calculus, we note that various cases of redundant codes in translation mentioned in 6.1 above often concern certain fixed ways of using names in processes, which would be amenable to type-based analysis.

Finally one of the most important topics which we could not touch in the present paper is the development of reasoning principles based on the proposed structuring method. Notice the type discipline in Section 5 already gives one simple, though useful, example. However it is yet to be studied whether reasoning methods on deeper properties of programs (cf. [1, 3, 20]) which are both mathematically well-founded and are applicable to conspicuous practical situations, can be developed or not. We wish to touch on this topic in our future study.

Acknowledgements. We sincerely thank three reviewers for their constructive comments. Discussions with Simon Gay, Luís Lopes, Benjamin Pierce, Kaku Takeuchi and Nobuko Yoshida have been beneficial, for which we are grateful. We also thank the last two for offering criticisms on an earlier version of the paper. Vasco Vasconcelos is partially supported by Project PRAXIS/2/2.1/MAT/46/94.

References

[1] S. Abramsky, S. Gay, and R. Nagarajan, Interaction Categories and Foundations of Typed Concurrent Computing. *Deductive Program Design*, Springer-Verlag, 1995.

[2] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986.

[3] P. America and F. de Boer. Reasoning about Dynamically Evolving Process Structures. *Formal Aspects of Computing*, 94:269–316, 1994.

[4] G. Berry and G. Boudol. The chemical abstract machine. *TCS*, 96:217–248, 1992.

[5] B. Courcelle. Fundamental properties of infinite trees. *TCS*, 25:95–169, 1983.

[6] C. Fournet and G. Gonthier. The reflexive chemical abstract machine and the join-calculus. In *POPL'96*, pp. 372–385, ACM Press, 1996.
[7] C.A.R. Hoare. Communicating sequential processes. *CACM*, 21(8):667–677, 1978.
[8] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1995.
[9] Jones, C.B., *Process-Algebraic Foundations for an Object-Based Design Notation*. UMCS-93-10-1, Computer Science Department, Manchester University, 1993.
[10] K. Honda. Composing processes. In *POPL'96*, pp. 344–357, ACM Press, 1996.
[11] K. Honda, V. Vasconcelos, and M. Kubo. Language primitives and type disciplines for structured communication-based programming. Full version of this paper, in preparation.
[12] S. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *POPL'96*, pp. 295–308, ACM Press, 1996.
[13] N. Kobayashi, B. Pierce, and D. Turner. Linearity and the pi-calculus. In *POPL'96*, pp. 358–371, ACM Press, 1996.
[14] L. Lopes, F. Silva, and V. Vasconcelos. *A framework for compiling object calculi*. In preparation.
[15] R. Milner. *Communication and Concurrency*. C.A.R. Hoare Series Editor. Prentice Hall, 1989.
[16] R. Milner. Functions as processes. *MSCS*, 2(2):119–141, 1992.
[17] R. Milner, Polyadic π-Calculus: a tutorial. *Logic and Algebra of Specification*, Springer-Verlag, 1992.
[18] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, Parts I and II. *Journal of Information and Computation*, 100:1–77, September 1992.
[19] R. Milner and M. Tofte. *The Definition of Standard ML*. MIT Press, 1991.
[20] H. R. Nielson and F. Nielson. Higher-order concurrent programs with finite communication topology. In *POPL '94*. ACM Press, 1994.
[21] A. Ohori and K. Kato. Semantics for communication primitives in a polymorphic language. In *POPL 93*. ACM Press, 1993.
[22] A. Ohori. A compilation method for ML-style polymorphic record calculi. In *POPL 92*, pp. 154–165. ACM Press, 1992.
[23] B. Pierce and D. Turner. Pict: A programming language based on the pi-calculus. CSCI Technical Report 476, Indiana University, March 1997.
[24] B. Pierce, and D. Sangiorgi, Typing and subtyping for mobile processes. In *LICS'93*, pp. 187–215, 1993.
[25] B. Pierce and D. Sangiorgi, Behavioral Equivalence in the Polymorphic Pi-Calculus. *POPL 97*, ACM Press, 1997.
[26] J-H. Reppy. CML: A higher-order concurrent language. In *PLDI 91*, pp. 293–305. ACM Press, 1991.
[27] K. Takeuchi, K. Honda, and M. Kubo. An interaction-based programming language and its typing system. In *PARLE'94*, volume 817 of *LNCS*, pp. 398–413. Springer-Verlag, July 1994.
[28] US Government Printing Office, Washington DC. *The Ada Programming Language*, 1983.
[29] V. Vasconcelos. Typed concurrent objects. In *ECOOP'94*, volume 821 of *LNCS*, pp. 100–117. Springer-Verlag, 1994.
[30] V. Vasconcelos and K. Honda, Principal Typing Scheme for Polyadic π-Calculus. *CONCUR'93*, Volume 715 of *LNCS*, pp.524-538, Springer-Verlag, 1993.
[31] V. Vasconcelos and M. Tokoro. A typing system for a calculus of objects. In *1st ISOTAS*, volume 742 of *LNCS*, pp. 460–474. Springer-Verlag, November 1993.
[32] Y. Yokote and M. Tokoro. Concurrent programming in ConcurrentSmalltalk. In Yonezawa and Tokoro [34].
[33] A. Yonezawa, editor. *ABCL, an Object-Oriented Concurrent System*. MIT Press, 1990.
[34] A. Yonezawa and M. Tokoro, editors. *Object-Oriented Concurrent Programming*. MIT Press, 1987.
[35] N. Yoshida, Graph Types for Monadic Mobile Processes. In *FST/TCS'16*, volume 1180 of *LNCS*, pp. 371–386, Springer-Verlag, 1996. The full version as LFCS Technical Report, ECS-LFCS-96-350, 1996.