Types for Dyadic Interaction

AUTHOR(S):

Honda, Kohei

CITATION:

Honda, Kohei. Types for Dyadic Interaction. 数理解析研究所講究録 1993, 851: 61-77

ISSUE DATE:

1993-10

URL:

http://hdl.handle.net/2433/83704

数理解析研究所講究録
第 851 巻 1993 年 61-77

Types for Dyadic Interaction*

Kohei Honda

kohei@mt.cs.keio.ac.jp

Department of Computer Science, Keio University
3-14-1 Hiyoshi, Kohoku-ku, Yokohama, 223, Japan

Abstract

We formulate a typed formalism for concurrency where types denote freely composable structure of dyadic interaction in the symmetric scheme. The resulting calculus is a typed reconstruction of name passing process calculi. Systems with both the explicit and implicit typing disciplines, where types form a simple hierarchy of types, are presented, which are proved to be in accordance with each other. A typed variant of bisimilarity is formulated and it is shown that typed β-equality has a clean embedding in the bisimilarity. Name reference structure induced by the simple hierarchy of types is studied, which fully characterises the typable terms in the set of untyped terms. It turns out that the name reference structure results in the deadlock-free property for a subset of terms with a certain regular structure, showing behavioural significance of the simple type discipline.

1 Introduction

This is a preliminary study of types for concurrency. Types here denote freely composable structure of dyadic interaction in the symmetric scheme. If we view concurrent computation as collection of interaction structures, types of computing naturally arise as mathematical encapsulation of those structures. Types are initially assigned to names. This assignment forms a universe of names, which is essentially Milner's sorting [18], and terms reside in this universe. Terms are formed from syntactic constructs corresponding to type structures, and equated by a variant of bisimilarity. The formalism can be regarded as a typed reconstruction of name passing process calculi [4, 15, 16, 7, 3, 12].

Types for interaction start from constant types, say **nat**, which denote fixed patterns of interaction. Then we have *input* and *output* types, symbolised as "$\downarrow\delta$" and "$\uparrow\delta$". Connotation of these primitives are to *receive* and *send* a value of type δ. That a type itself occurs in a type is reminiscent of functional types. We also have a *unit type*, "**1**", denoting the *inaction*. These types are now combined to become a composite type by connectives of *sequencing* and *branching*. Sequencing composes two types sequentially and is denoted by ";", hence we have a structure such as "$\delta_1;\delta_2$", which means *first* to engage in an action of type δ_1 and *then* engage in an action of type δ_2. In branching we have two constructors. One is "&", with which we have a structure such as "$\delta_1\&\delta_2$" where one offers two alternatives to wait for one of them to be selected. Another is "$\delta_1 \oplus \delta_2$," which selects the right option or the left option of a "&" type. Operationally a composite type denotes semantically atomic operation which in fact is an amalgamation of multiple interactive communication. As a whole, the set of types form a simple hierarchy, reminiscent of Church's simple hierarchy of functional types.

We then form a typed universe of names, where each name corresponds to a certain type. Then *terms* and *actions* are formed from names. Terms are made up of actions, each of which corresponds to a certain type and substantiates the abstract type structure operationally. Actions provide high-level abstraction for interactive behaviour, just as λ-abstraction, arrays, and records provides abstraction for their respective operations in the sequential setting; they are *concrete type constructors* for concurrent programming. Transition semantics is formulated and a typed variant of bisimilarity equate terms. It is shown, via Milner's encoding of lazy λ-calculus, that the explicitly typed β-equality can be embedded in explicitly typed weak bisimilarity.

*An extended version of [11].

Pragmatically it is important to be able to reconstruct typed terms from untyped terms. In the present scheme, a simple type inference system suffices, which is essentially an extension of the foregoing construction for functional types. The shape of the typing judgement, however, is notably different from the foregoing one. The sequent takes a form:

$$\vdash P \succ x_1 : \alpha_1,\ x_2 : \alpha_2, ...,\ x_n : \alpha_n$$

where P is a term, x_i are port names, α_i's are type schemes. Basically the judgement tells us the structures of potential interface points a term owns. The inference system enjoys Subject Reduction and existence of Principal Typing, and assigns types exactly, modulo α-equality, to the set of terms which are the result of erasing type annotations from explicitly typed terms. One essential result is that a name reference structure in terms precisely differentiates typable terms from untypable terms. It is exhibited that, in a certain set of terms, the name reference structure induced by the simple type hierarchy results in the *deadlock-free behaviour*, one of important properties in concurrent or distributed computing. The result would lay a foundation for further investigation of varied type disciplines in the name passing framework.

The structure of the rest of the paper follows. The explicitly typed system is studied in Sections 2 and 3. Section 2 introduces types and typed terms, then defines reduction relation over terms. Section 3 first defines typed bisimilarity, then presents the embedding result of the explicitly typed β-equality in the bisimilarity. Section 4 then introduces the implicitly typed system. After basic definitions, syntactic properties of the type inference system are presented and correspondence with the explicitly typed system is established. Section 5 studies how the present "simple" type discipline is reflected in syntactic and behavioural properties of terms. We give a complete characterization of typable terms by a name reference structure, and show that the typability induces deadlock free property for a certain subset of terms. Section 6 discusses related work and further issues.

All proofs are omitted for the space sake. Interested readers may consult [10] for them.

2 Typed Terms

2.1. Types and co-types. Initially there are *types*. Symmetry intrinsic in interaction is reflected by the syntax of types — if we have a type of natural numbers (write it $\mathtt{nat}$), then we have a type of co-natural numbers, $\overline{\mathtt{nat}}$. Let us have the set of atomic types, denoted $\mathbf{At}$, ranged over by $C, C', \ldots$. Then to each atomic type there exists another atomic type, called its *co-type*. The co-type of the co-type of an atomic type is always the original type.

We assume that $\mathbf{At}$ at least includes $\mathtt{nat}$, $\overline{\mathtt{nat}}$, and $\mathbf{1}$. Intuitively $\mathtt{nat}$ is a type of natural number, while $\overline{\mathtt{nat}}$ is a type which interacts with it (like *successors*). $\overline{\mathtt{nat}}$ is the co-type of $\mathtt{nat}$. $\mathbf{1}$ is rather special – it denotes a pure form of synchronization (and resulting term generation) in the line of, say, CCS. Its co-type is itself. How $\mathbf{1}$ acts in the concrete examples, see examples in 2.5 later. We call atomic types which are not $\mathbf{1}$, *constant types*.

The set of types for interaction, $\mathbf{T}$, ranged over by $\delta, \delta', \ldots$, is defined as follows.

(i) An atomic type is a type.

(ii) If δ is a type, then $\downarrow\delta$ and $\uparrow\delta$ are types.

(iii) If δ_1 and δ_2 are types, then $\delta_1;\delta_2$, $\delta_1 \& \delta_2$ and $\delta_1 \oplus \delta_2$ are types.

Intuitively $\downarrow\delta$ denotes a type which receives δ, while $\uparrow\delta$ denotes a type which emits δ. $\delta_1;\delta_2$ denotes sequential composition of two types, $\delta_1 \& \delta_2$ denotes a type who offers two alternatives, and $\delta_1 \oplus \delta_2$ denotes a type which would select left or right and interact by the according type.

The notion of co-types is extended to composite types. The co-type of a type δ is denoted $\overline{\delta}$, implying a kind of an "interaction pair" of a type δ. The notion is to be substantiated computationally later.

(i) $\overline{C}$ is the co-type of C.

(ii) $\overline{\downarrow\delta} \stackrel{\text{def}}{=} \uparrow\overline{\delta}$, and $\overline{\uparrow\delta} \stackrel{\text{def}}{=} \downarrow\overline{\delta}$.

(iii) $\overline{\delta;\delta'} \stackrel{\text{def}}{=} \overline{\delta};\overline{\delta'}$, $\overline{\delta\&\delta'} \stackrel{\text{def}}{=} \overline{\delta} \oplus \overline{\delta'}$, and $\overline{\delta \oplus \delta'} \stackrel{\text{def}}{=} \overline{\delta}\&\overline{\delta'}$.

It is easy to know:

PROPOSITION 2.1 $\overline{\overline{\delta}} \stackrel{\text{def}}{=} \delta$ *for any type* δ.

Some syntactic conventions: we assume the strength of association, from the strongest, is as follows. $\{\downarrow,\uparrow\}$, $\{;\}$, and $\{\&,\oplus\}$. We also assume all constructors associate to the right.

2. 2. Names, terms and actions. In each type a set of names reside. The mapping from types to typed names is called *universe*. Let a total function $\mathcal{U}$ be such that it assigns, to each type δ, a denumerable set of names, written $a^\delta, b^\delta, c^\delta..$, or $x^\delta, y^\delta, z^\delta...$ Sometimes we omit type annotations, and write $a, b, c, ..$ or $x, y, z, ...$ Each universe is equipped with an irreflexive, bijective mapping from names to names, written "$\overline{a^\delta}$", where we should have $a^\delta \in \mathcal{U}(\delta) \Leftrightarrow \overline{a^\delta} \in \mathcal{U}(\overline{\delta})$. Thus the names in the type δ should be exactly the co-names of the names in the type $\overline{\delta}$. We often write $\overline{a}^{\overline{\delta}}$ or simply $\overline{a}$ for $\overline{a^\delta}$.

Since any universe is isomorphic in structure, we will often fix some universe and discuss within it. Let the concerned universe be $\mathcal{U}$. Then we form typed expressions there. In functional types we only have one entity, called *terms*. Here we have – other than *names* – two entities, *actions* and *terms*. An action denotes a semantically atomic structure of interaction, and corresponds to one type. A term denotes a structured collection of independent actions. What essentially differs from the situation in functional types is that a term does *not* inhabit a particular type. At best, terms only inhabit the underlying universe.

Let us assume that we have a set of *constant symbols* corresponding to each constant type (that is, atomic types except 1). Each constant symbol has its *arity* of the form $[\delta_1 \ldots \delta_n]$ with $n \geq 0$. Arity denotes types of names the constant would carry.

The sets of terms and actions in a universe are defined inductively. $\mathcal{V}^\delta, \mathcal{V}'^{\delta'}, \ldots$ range over typed actions[1], while $P, Q, R, \ldots$ range over typed terms. Note it is implicit that the underlying universe is $\mathcal{U}$; "a term" and "an action" mean "a term in $\mathcal{U}$" and "an action in $\mathcal{U}$", respectively.

(i) If c is a constant symbol of type C and of arity $[\delta_1 \ldots \delta_n]$, and $x_i^{\delta_i}$ is a name $(i = 1..n)$, then $\mathsf{c}(x_1{}^{\delta_1} \ldots x_n{}^{\delta_n})^C$ is an action, called *constant action*.

(ii) If x^δ is a name, $(\triangleright x^\delta)^{\downarrow\delta}$ and $(\triangleleft x^\delta)^{\uparrow\delta}$ are actions.

(iii) If $\mathcal{V}_1{}^{\delta_1}$ and $\mathcal{V}_2{}^{\delta_2}$ are actions, $(\mathcal{V}_1{}^{\delta_1};\mathcal{V}_2{}^{\delta_2})^{\delta_1;\delta_2}$ and $([\mathcal{V}_1{}^{\delta_1}]\&[\mathcal{V}_2{}^{\delta_2}])^{\delta_1\&\delta_2}$ are actions.

(iv) If $\mathcal{V}^\delta$ is an action and δ' is a type, $\mathsf{inl}(\mathcal{V}^\delta)^{\delta\oplus\delta'}$ and $\mathsf{inr}(\mathcal{V}^\delta)^{\delta'\oplus\delta}$ are actions.

(v) If P is a term, $(.P)^{\mathbf{1}}$ is an action.

(vi) If $\mathcal{V}^\delta$ is an action and a^δ is a name, $a^\delta : \mathcal{V}^\delta$ is a term. This is a *prime term* and a^δ is its *subject*.

(vii) If P and Q are terms, $(P.Q)$ is a term. This is a *parallel composition* of P and Q.

(viii) If P is a term, $|x^\delta|P$ is a term. This is a *scope restriction* of x^δ in P.

(ix) Λ is a term. Λ is an *inaction*.

(x) If P is a term, $!P$ is a term. This is a *replication* of P.

Some conventions: we assume all binary constructors associate to right, though we still use parenthesis to be explicit about syntactic structure. Since it is sometimes cumbersome to write typed expressions in full, e.g.

$$((\triangleright a^{\mathsf{nat}})^{\downarrow\mathsf{nat}};(\triangleleft b^{\overline{\mathsf{nat}}})^{\uparrow\overline{\mathsf{nat}}})^{\downarrow\mathsf{nat};\uparrow\overline{\mathsf{nat}}}$$

[1] An alert reader may perceive the resemblance between Milner's syntactic construction in [18]. In fact his notion of "agents" may be regarded as a sprout of the syntactic category which we now call "actions". See also his suggestion to "intermingle abstraction and concretion" in [18]. We note that a calculus in [9] (complete in October 1991) already contained the concrete type structures corresponding to actions, and we came to know [18] in February 1992. The coincidence of ideas, however, is quite notable.

we will abbreviate internal type script and write:

$$\triangleright a;\triangleleft b^{\downarrow \mathbf{nat};\uparrow\overline{\mathbf{nat}}}$$

from which the reconstruction of the original expression is easy. In some cases, where type annotations are not necessary, we will even write this "$\triangleright a;\triangleleft b$", but it is always assumed that the expression is essentially typed. Another simplification of syntax is to write e.g. $\triangleleft a \triangleright x$ for $\triangleleft a; \triangleright x^{\uparrow\delta_1;\downarrow\delta_2}$ (i.e. omitting ";"), or even to write e.g. $\triangleleft a v \triangleright x^{\uparrow\delta_1;\uparrow\delta_2;\downarrow\delta_2}$ for $\triangleleft a;\triangleleft v;\triangleright x^{\uparrow\delta_1;\uparrow\delta_2;\downarrow\delta_3}$ (i.e. omitting $\triangleleft$ and $\triangleright$ symbols in consecutive inputs and outputs), which is often convenient.

2. 3. Binding and substitution. Bindings are induced by two binders. In (2) we omit type scripts.

(1) In $|x^\delta|P$, x^δ binds free occurrences of x^δ and $\overline{x}^{\overline{\delta}}$ in P.

(2) We say the occurrence of x in $\triangleright x$ is effective. If x is effective in $\mathcal{V}$ then it is also effective in $[\mathcal{V}]\&[\mathcal{V}']$, $[\mathcal{V}']\&[\mathcal{V}]$, $\mathsf{inl}(\mathcal{V})$, $\mathsf{inr}(\mathcal{V})$, and $\mathcal{V};\mathcal{V}'$, except in the last case when another x is effective in $\mathcal{V}'$. Now if an occurrence of x is effective in $\mathcal{V}$, then it binds any free x in $\mathcal{V}'$ within $\mathcal{V};\mathcal{V}'$.

With respect to (2) above, we will henceforth assume that in $[\mathcal{V}]\&[\mathcal{V}']$, if a name x is effective in $\mathcal{V}$ (resp. $\mathcal{V}'$) then it should be also effective in $\mathcal{V}'$ (resp. $\mathcal{V}$). To understand the intuition behind (2), think of $([\triangleleft v;\triangleright x]\&[\triangleleft u;\triangleleft w;\triangleright x]);\triangleleft x$. or $\mathsf{inl}(\triangleleft v;\triangleright x);\triangleleft x$. In both it is natural to assume that x in $\triangleleft x$ is bound. Moreover the binding notion is in harmony with interesting equality over types such as $\delta;\mathbf{1} = \mathbf{1};\delta = \delta$ and corresponding equivalence over actions, though we do not stipulate them in this expository paper. For the sake of simplicity, the following convention is convenient without losing generality.

CONVENTION 2.2 *We assume all names in binding occurrences in any typed expressions are pairwise distinct and disjoint from free names.*

Now we define (typed) substitution of names. A free name with a certain type is substituted for a name with the same type. A substitution of a name for another name is effective also to their co-names, which is in harmony with the binding idea, so that the definition of typed substitution starts from:

$$x^\delta[v^\delta/x^\delta] \stackrel{\text{def}}{=} x^\delta[\overline{v}^{\overline{\delta}}/\overline{x}^{\overline{\delta}}] \stackrel{\text{def}}{=} v^\delta \qquad \overline{x}^{\overline{\delta}}[v^\delta/x^\delta] \stackrel{\text{def}}{=} \overline{x}^{\overline{\delta}}[\overline{v}^{\overline{\delta}}/\overline{x}^{\overline{\delta}}] \stackrel{\text{def}}{=} \overline{v}^{\overline{\delta}}$$

the rest being the standard. Since there are infinitely many names in each type, we can always find a fresh name for each type, and this makes substitution well-defined. We will mainly use simultaneous substitutions, written $\{\tilde{v}^{\tilde{\delta}}/\tilde{x}^{\tilde{\delta}}\}$. We will omit type superscripts as in expressions, but even when we do so, we always assume that the substitution is well-typed. σ_{id} denotes the identity operation. *Sequential composition* of σ and σ', written $\sigma;\sigma'$, denotes the result of firstly performing σ as far as the substituted names do not collide with those of σ', which means, under Convention 2.2, simply $P(\sigma;\sigma') \stackrel{\text{def}}{=} (P\sigma)\sigma'$. The following is immediate from the definition of substitution.

PROPOSITION 2.3 *If P is a term then $P[v^\delta/x^\delta]$ is also a term. Similarly, if $\mathcal{V}^\delta$ is an action, then $\mathcal{V}[v^{\delta'}/x^{\delta'}]$ is also an action.*

So substitution is a well-typed operation. Then we define α-convertibility, written $\equiv_\alpha$, in the standard way. Substitution and α-convertibility are naturally restricted to those terms under Convention 2.2.

2. 4. Reduction (1). Reduction represents the fundamental mechanism of computing in our formalism. Since an action is composition of several fine-grained operations in general, we need to decompose the definition of reduction rules into that for actions and that for terms. The reduction for actions: $\mathcal{V} \odot \mathcal{V}' \rightsquigarrow \sigma \cdot \sigma' \cdot P$ means that, when $\mathcal{V}$ interacts with $\mathcal{V}'$, one term and two substitutions are generated. Substitutions are used to make the receipt of the values effective in actions sequentially composed later. The notion is consistent with the binding idea discussed in 2.3. The definition

of reduction starts from how two constants interact together. Let us assume that, for each pair of constant actions with dual types, we have a rule in the form:

$$\text{(const)} \quad \mathsf{c}(\tilde{x})^{C} \odot \mathsf{c}'(\tilde{y})^{\overline{C}} \rightsquigarrow \sigma_{id} \cdot \sigma_{id} \cdot P$$

where all names are distinct and $\mathcal{FN}(P) \in \{\tilde{x}\tilde{y}\}$. One example is

$$N^{\mathtt{nat}} \odot succ(x^{\delta})^{\overline{\mathtt{nat}}} \rightsquigarrow \sigma_{id} \cdot \sigma_{id} \cdot !x : N'^{\mathtt{nat}}$$

where N' denotes the successor to N. The set of rules for reduction of actions follow.

$$\text{(pass)} \quad \lhd v^{\delta} \odot \rhd \overline{x}^{\overline{\delta}} \rightsquigarrow \sigma_{id} \cdot [v/x] \cdot \Lambda \qquad \text{(gen)} \quad .P^{\mathbf{1}} \odot .Q^{\mathbf{1}} \rightsquigarrow \sigma_{id} \cdot \sigma_{id} \cdot (P, Q)$$

$$\text{(inl)} \quad \frac{\mathcal{V}_1{}^{\delta_1} \odot \mathcal{V}^{\overline{\delta_1}} \rightsquigarrow \sigma_1 \cdot \sigma' \cdot P_1}{\mathcal{V}_1 \& \mathcal{V}_2{}^{\delta_1 \& \delta_2} \odot \mathsf{inl}(\mathcal{V})^{\overline{\delta_1 \oplus \delta_2}} \rightsquigarrow \sigma_1 \cdot \sigma \cdot P_1} \qquad \text{(inr)} \quad \frac{\mathcal{V}_2{}^{\delta_2} \odot \mathcal{V}^{\overline{\delta_2}} \rightsquigarrow \sigma_2 \cdot \sigma \cdot P_2}{\mathcal{V}_1 \& \mathcal{V}_2{}^{\delta_1 \& \delta_2} \odot \mathsf{inr}(\mathcal{V})^{\overline{\delta_1 \oplus \delta_2}} \rightsquigarrow \sigma_2 \cdot \sigma \cdot P_2}$$

$$\text{(seq)} \quad \frac{\mathcal{V}_1{}^{\delta_1} \odot \mathcal{V}_1'{}^{\overline{\delta_1}} \rightsquigarrow \sigma_1 \cdot \sigma_1' \cdot P \quad \mathcal{V}_2 \sigma_1{}^{\delta_2} \odot \mathcal{V}_2' \sigma_1'{}^{\overline{\delta_2}} \rightsquigarrow \sigma_2 \cdot \sigma_2' \cdot Q}{\mathcal{V}_1; \mathcal{V}_2{}^{\delta} \odot \mathcal{V}_1'; \mathcal{V}_2'{}^{\overline{\delta}} \rightsquigarrow \sigma_1; \sigma_2 \cdot \sigma_1'; \sigma_2' \cdot (P, Q)} \qquad \text{(exc)} \quad \frac{\mathcal{V}_1{}^{\delta} \odot \mathcal{V}_2{}^{\overline{\delta}} \rightsquigarrow \sigma_1 \cdot \sigma_2 \cdot P}{\mathcal{V}_2{}^{\delta} \odot \mathcal{V}_1{}^{\delta} \rightsquigarrow \sigma_2 \cdot \sigma_1 \cdot P}$$

Note that reduction rules are provided only for a pair of actions with dual types. Since we do not need the substitution for top level actions (i.e. actions which occur in a term without being composed with another), we will sometimes write $\mathcal{V}_1{}^{\delta} \odot \mathcal{V}_2{}^{\overline{\delta}} \rightsquigarrow P$, omitting substitutions. Easily we get:

PROPOSITION 2.4 *Given two actions $\mathcal{V}_1{}^{\delta}$ and $\mathcal{V}_2{}^{\overline{\delta}}$, there is always a unique P such that $\mathcal{V}_1{}^{\delta} \odot \mathcal{V}_2{}^{\overline{\delta}} \rightsquigarrow P$.*

2.5. Reduction (2). We next define structural rules over terms, used to define reduction relation. The structural congruence, denoted $\equiv$, is the smallest congruence relation induced by the following rules.

(1)	$P \equiv Q$ (if $P \equiv_{\alpha} Q$)	(4)	$\lvert x\rvert P, Q \equiv \lvert x\rvert(P, Q)$ $(x, \overline{x} \notin \mathcal{FN}(Q))$
(2)	$(P, Q), R \equiv P, (Q, R)$	(5)	$P, Q \equiv Q, P$
(3)	$P, \Lambda \equiv P$	(6)	$!P \equiv P, !P$

Let us write the sequence of concurrent composition of prime terms and replications, ∂, ∂', etcetera. The main definition follows.

DEFINITION 2.5 *One-step reduction, denoted $\longrightarrow$, is the smallest relation over terms generated by:*

$$\text{(com)} \quad \frac{\mathcal{V}_1{}^{\delta} \odot \mathcal{V}_2{}^{\overline{\delta}} \rightsquigarrow P}{\lvert\tilde{w}\rvert(\partial,\ a : \mathcal{V}_1{}^{\delta},\ \overline{a} : \mathcal{V}_2{}^{\overline{\delta}},\ \partial') \longrightarrow \lvert\tilde{w}\rvert(\partial,\ P,\ \partial')} \qquad \text{(struct)} \quad \frac{P_1' \equiv P_1 \quad P_1 \longrightarrow P_2 \quad P_2 \equiv P_2'}{P_1' \longrightarrow P_2'}$$

We also define $\longrightarrow\!\!\!\!\rightarrow$ as $\rightarrow^{*} \cup \equiv$. By Proposition 2.3, we immediately have:

THEOREM 2.6 *If P is a term and $P \longrightarrow P'$, then P' is also a term (in the same universe).*

Some examples of terms and their reduction follow.

EXAMPLE 2.7

(i) *Constant.* Let the arity of $succ(x)$ be $[\mathtt{nat}]$ and assume the rule which has been already mentioned in 2.4. Then:

$$(!a : 3^{\mathtt{nat}},\ \overline{a} : succ(x)^{\overline{\mathtt{nat}}}) \longrightarrow (!a : 3^{\mathtt{nat}},\ x : 4^{\mathtt{nat}})$$

(ii) *Bi-directional interaction.* Note v comes back to the agent on the left. If one allows only one-directional value passing, the result is what we can see in Polyadic π-calculus [18].

$$(a : \lhd v; \lhd w; \rhd \overline{x}; .P,\ \overline{a} : \rhd \overline{x}; \rhd \overline{y}; \lhd x; .Q) \longrightarrow (P\{v/x\},\ Q\{vw/xy\})$$

(iii) *Branching and term generation.* Let us assume the following terms.

$$\begin{array}{llll} true(b) & \stackrel{\text{def}}{=} & b : \mathsf{inl}(.\Lambda)^{\mathbf{1}\oplus\mathbf{1}} & dupl(bx) \stackrel{\text{def}}{=} b : [.true(x)]\&[.false(x)]^{\mathbf{1}\&\mathbf{1}} \\ false(b) & \stackrel{\text{def}}{=} & b : \mathsf{inr}(.\Lambda)^{\mathbf{1}\oplus\mathbf{1}} & and(b_1b_2z) \stackrel{\text{def}}{=} b_1 : [.dupl(b_2z)]\&[.false(z)]^{\mathbf{1}\&\mathbf{1}} \\ not(b_1b) & \stackrel{\text{def}}{=} & b_1 : [.false(b)]\&[.true(b)]^{\mathbf{1}\&\mathbf{1}} & or(b_1b_2z) \stackrel{\text{def}}{=} b_1 : [.true(z)]\&[.dupl(b_2z)]^{\mathbf{1}\&\mathbf{1}} \end{array}$$

Then we have, for example:

$$(true(x),\ false(y),\ or(\overline{xy}z)) \longrightarrow (true(x),\ false(y),\ true(z))$$

Note we can define *parallel or* by: $por(xyz) \stackrel{\text{def}}{=} (or(xyz), or(yxz))$.

(iv) Buffer (1). $buffer(a : x_1..x_n)$ denotes a buffer with the oldest value x_1 and the newest value x_n, with the interaction port a. The left branch is for "read" request, while the right branch responds to the "write" request. If it is empty, it would tell that to the user. Note how exception handling is elegantly embedded in the branching structure.

$$\begin{array}{l} buffer(a : \varepsilon) \stackrel{\text{def}}{=} \\ \quad a : [\mathsf{inl}(.buffer_0(a : \varepsilon))]\&[\triangleright x.buffer(a : x)] \\ buffer(a : v\tilde{x}) \stackrel{\text{def}}{=} \\ \quad a : [\mathsf{inr}(\triangleleft w.buffer(a : \tilde{x}))]\&[\triangleright w.buffer(a : v\tilde{x}w)] \end{array}$$

(v) Buffer (2). We define the same behaviour as amalgamation of small parts. We can verify they are in fact behaviourally equivalent using the typed bisimilarity later. We first define two parts, an empty cell and a usual cell.

$$\begin{array}{l} empty(a) \stackrel{\text{def}}{=} \\ \quad a : [\mathsf{inl}(.empty(a))]\&[\triangleright x.|n|(cell(axn), empty(n))] \\ cell(avn) \stackrel{\text{def}}{=} \\ \quad a : [\mathsf{inr}(\triangleleft v.\overline{n} : \mathsf{inl}([.empty(a)]\&[\triangleright y.cell(ayn)]))]\ \& \\ \qquad [\triangleright y.(\overline{n} : \mathsf{inr}(\triangleleft y, cell(avn)))] \end{array}$$

The new buffer is defined as:

$$\begin{array}{lll} newbuffer(a : \varepsilon) & \stackrel{\text{def}}{=} & empty(a) \\ newbuffer(a : v\tilde{x}) & \stackrel{\text{def}}{=} & |n|(cell(avn), newbuffer(n : \tilde{x})) \end{array}$$

In the last three examples, "&" (as an action constructor) roughly plays a role of usual summation in process calculi, i.e. to express branching of behaviour. One essential difference lies in that type abstraction for summation may not be possible. This is related with *local* behaviour of the branching construct. At the same time, we cannot transform concurrent composition into & (as in so-called "expansion law"), which is another crucial difference from the general summation.

3 Transition and Bisimulation

3. 1. Labels and Instantiation. We introduce labelled transition relation in the following, to define usual bisimilarity for processes in the typed setting[2]. As in reduction, one transition may represent an amalgamation of many small interactions.

[2]While it is possible to define behavioural semantics based on reduction relation, that is, without relying on labelled transition, as in [12, 17], we do not do so in this paper, since these methods are fairly new, and labelled transition relation in the typed setting itself is an important subject of study. We note that the semantics we develop in this section conforms to both of the equivalences induced by two means developed in [12, 17].

There are two kinds of labels: one is τ, which means that interaction takes place within a term and coincides with reduction. Another takes a form $a : \mathcal{E}$, where $\mathcal{E}$ is called an *action instantiation.* It denotes an interaction with the outside a term engages in. The grammar to define the set of action instantiations follows[3].

$$\mathcal{E} ::= * \mid \mathsf{c}(\tilde{x}) \mid \uparrow x \mid \downarrow x \mid \mathcal{E}_1; \mathcal{E}_2 \mid [\mathcal{E}]\& \mid \&[\mathcal{E}] \mid \mathsf{inl}(\mathcal{E}) \mid \mathsf{inr}(\mathcal{E})$$

where $\mathsf{c}(\tilde{x})^C$ is a constant action. The relationship between the action expression and these instantiations is given by the notation:

$$\mathcal{V}^\delta \overset{\mathcal{E}}{\rightsquigarrow} \sigma \cdot P$$

which reads "$\mathcal{V}$ is instantiated into $\mathcal{E}$ and generates σ and P". We first assume that the rule (const) exists for each pair of two constant symbols with dual types[4].

$$(\text{const}) \ \frac{\mathsf{c}(\tilde{x})^C \odot \mathsf{c}'(\tilde{y})^{\overline{C}} \rightsquigarrow P}{\mathsf{c}(\tilde{x})^C \overset{\mathsf{c}(\tilde{x}) \cdot \mathsf{c}'(\tilde{y})}{\rightsquigarrow} \sigma_{id} \cdot P}$$

For other actions we have the following rules.

$$(\text{in}) \ (\triangleright x^\delta)^{\downarrow\delta} \overset{\downarrow v^\delta}{\rightsquigarrow} \{v/x\} \cdot \Lambda \qquad (\text{out}) \ (\triangleleft v^\delta)^{\uparrow\delta} \overset{\uparrow v^\delta}{\rightsquigarrow} \Lambda \cdot \sigma_{id} \qquad (\text{gen}) \ .P^{\mathbf{1}} \overset{*}{\rightsquigarrow} \sigma_{id} \cdot P$$

$$(\text{seq}) \ \frac{\mathcal{V}_1{}^{\delta_1} \overset{\mathcal{E}_1}{\rightsquigarrow} \sigma_1 \cdot P_1 \qquad \mathcal{V}_2\sigma_1{}^{\delta_2} \overset{\mathcal{E}_2}{\rightsquigarrow} \sigma_2 \cdot P_2}{\mathcal{V}_1; \mathcal{V}_2{}^{\delta_1;\delta_2} \overset{\mathcal{E}_1;\mathcal{E}_2}{\rightsquigarrow} \sigma_1;\sigma_2 \cdot (P, Q)} \qquad (\text{with(l)}) \ \frac{\mathcal{V}^\delta \overset{\mathcal{E}}{\rightsquigarrow} \sigma \cdot P}{\mathcal{V}\&\mathcal{V}'^{\delta\&\delta'} \overset{[\mathcal{E}]\&}{\rightsquigarrow} \sigma \cdot P}$$

$$(\text{with(r)}) \ \frac{\mathcal{V}'^{\delta'} \overset{\mathcal{E}'}{\rightsquigarrow} \sigma' \cdot P'}{\mathcal{V}\&\mathcal{V}'^{\delta\&\delta'} \overset{\&[\mathcal{E}']}{\rightsquigarrow} \sigma' \cdot P'} \qquad (\text{plus(l)}) \ \frac{\mathcal{V}^\delta \overset{\mathcal{E}}{\rightsquigarrow} P \cdot \sigma}{\mathsf{inl}(\mathcal{V})^{\delta\oplus\delta'} \overset{\mathsf{inl}(\mathcal{E})}{\rightsquigarrow} P \cdot \sigma} \qquad (\text{plus(r)}) \ \frac{\mathcal{V}'^{\delta'} \overset{\mathcal{E}'}{\rightsquigarrow} \sigma' \cdot P'}{\mathsf{inr}(\mathcal{V}')^{\delta\oplus\delta'} \overset{\mathsf{inr}(\mathcal{E}')}{\rightsquigarrow} \sigma' \cdot P'}$$

As in the reduction relation, for top level actions we write $\mathcal{V}^\delta \overset{\mathcal{E}}{\rightsquigarrow} P$ omitting σ. The following is proved by induction on the inference rules presented above.

PROPOSITION 3.1 *Suppose* $\mathcal{V}^\delta \overset{\mathcal{E}}{\rightsquigarrow} \sigma \cdot P$. *Then* σ *is well-typed and* P *is a term.*

3.2. The transition system and bisimilarity. Now we define transition relation $\xrightarrow{l}$, whose element is in the form of $\langle P, l, Q\rangle$, where P and Q are typed terms in a certain universe and l is a label, written $P \xrightarrow{l} Q$, as follows. Z

$$(\text{inter}) \ \frac{\mathcal{V}^\delta \overset{\mathcal{E}}{\rightsquigarrow} P}{|\tilde{w}|(\partial, a : \mathcal{V}^\delta, \partial') \xrightarrow{a:(\tilde{x})\mathcal{E}} |\tilde{w}\backslash\tilde{x}|(\partial, P, \partial')} \qquad (\text{com}) \ \frac{\mathcal{V}_1{}^\delta \odot \mathcal{V}_2{}^{\overline{\delta}} \rightsquigarrow P}{|\tilde{w}|(\partial, \ a : \mathcal{V}_1{}^\delta, \ a : \mathcal{V}_2{}^{\overline{\delta}}, \ \partial') \xrightarrow{\tau} |\tilde{w}|(\partial, \ P, \ \partial')}$$

$$(\text{struct}) \ \frac{P_1' \equiv P_1 \quad P_1 \xrightarrow{l} P_2 \quad P_2 \equiv P_2'}{P_1' \xrightarrow{l} P_2'}$$

where, in (inter), we assume $\{\tilde{x}\} \in \{\tilde{w}\} \cap \mathcal{FN}(\mathcal{E})$. Here names in $(\tilde{x})$ are binding occurrences. Now the following tells us that the transition rules above are essentially typed; the result is immediate from Proposition 3.1.

THEOREM 3.2 *If* $P \xrightarrow{l} P'$ *and* P *is a term, then* P' *is also a term in the same universe.*

[3] We note that, if we equate e.g. $\mathcal{V}; .\Lambda$ or $.\Lambda; \mathcal{V}$ with $\mathcal{V}$, hence $\delta; \mathbf{1} = \mathbf{1}; \delta = \delta$, as suggested in 2.3, we can have a more abstract notion of labels, i.e. $*; \mathcal{E} = \mathcal{E}; * = \mathcal{E}$, which in turn is reflected on bisimilarity to induce a more general but still sound semantic relation. For simplicity, this path is not pursued here.

[4] In the rule the generation of the whole term P is sound since τ-transition is defined without labelled transition.

Thus, in our formalism, all terms which reside in a universe would always compute and interact within that universe.

We now formulate typed weak bisimilarity based on the labelled transition relation. Below $\stackrel{\hat{l}}{\Longrightarrow}$ is defined as $\longrightarrow\!\!\!\!\!\rightarrow$ if $l = \tau$, else as $\longrightarrow\!\!\!\!\!\rightarrow \stackrel{l}{\longrightarrow} \longrightarrow\!\!\!\!\!\rightarrow$.

DEFINITION 3.3 *P_1 and Q_1 are weakly bisimilar (in the concerned universe), denoted by $P_1 \approx Q_1$, if there exists a symmetric relation $\mathcal{R}$ over terms with $\langle P_1, Q_1\rangle \in \mathcal{R}$, such that, for any $\langle P, Q\rangle \in \mathcal{R}$ we have, whenever $P \stackrel{l}{\longrightarrow} P'$ where $\mathcal{BN}(l)$ does not occur free in Q, for some Q', $Q \stackrel{\hat{l}}{\Longrightarrow} Q'$ and $\langle P', Q'\rangle \in \mathcal{R}$.*

The notion is well-defined by Theorem 3.2. As to the bisimilarity, we have the following. The proof basically proceeds as in usual name passing calculi (see [15, 9, 12]) and is omitted.

THEOREM 3.4 *$\approx$ is a congruence relation.*

Note the notion of congruence here is in terms of typed formation rules given in 2.2. With the theorem, we can regard $\approx$ as providing the equality notion for typed processes.

3.3. Embedding simply typed β-theory. As an example application of typed equality by $\approx$, let us see how the typed β-theory over typed λ-terms can be embedded in $\approx$, using Milner's encoding of lazy λ-calculus [16]. We leave the definition of a simply typed λ-calculus to e.g. [2]. Naturally we take the explicitly typed system. For simplicity only one constant type $\mathtt{NAT}$ is assumed and we write $\alpha \to \beta$ to denote the functional type whose domain is α and co-domain is β. Term formation starts from variables and constants (e.g. natural numbers and successor function)[5], and written e.g. M^α, $(\lambda x^\alpha.M^\beta)^{\alpha\to\beta}$, etcetera. First we assume the following embedding of types:

(i) $\mathtt{NAT}^\bullet \stackrel{\text{def}}{=} \mathtt{nat}$.

(ii) $(\alpha \to \beta)^\bullet \stackrel{\text{def}}{=} \downarrow(\downarrow\alpha^\bullet); \downarrow\overline{\beta^\bullet}; \mathbf{1}$,

Note how vital is the existence of constant types in this mapping. We suspect that the reason that the similar construction has not been done in the sorting discipline is precisely because they lack the notion of constant types (and constant actions, for that matter) in their framework. Now it is clear, for each functional type α, there exists its image $\alpha^\bullet$ in $\mathbf{T}$. To conform to the structure of the encoding, we assume a universe in which, for each functional variable x^α, we have $x : \downarrow\alpha^\bullet$. The mapping follows. Actually the mapping maps the tuple $\langle M^\alpha, u^{\alpha^\bullet}\rangle$ to a certain term in the universe. Below we use the name u throughout but it should be thought to denote some name with an appropriate type.

(i) $[\![\mathrm{N}^{\mathtt{NAT}}]\!]u \stackrel{\text{def}}{=} u : \mathrm{N}^{\mathtt{NAT}^\bullet}$ and $[\![succ^{\mathtt{NAT}\to\mathtt{NAT}}]\!]u \stackrel{\text{def}}{=} u : \rhd\overline{xy}.x : succ(y)^{(\mathtt{NAT}\to\mathtt{NAT})^\bullet}$.

(ii) $[\![x^\alpha]\!]u \stackrel{\text{def}}{=} \overline{x} : \lhd u^{\lceil\alpha^\bullet}$.

(iii) $[\![(\lambda x^\alpha.M^\beta)^{\alpha\to\beta}]\!]u \stackrel{\text{def}}{=} u : (\rhd\overline{xy}.[\![M^\beta]\!]y)^{(\alpha\to\beta)^\bullet}$.

(iv) $[\![(M^{\alpha\to\beta}N^\alpha)^\beta]\!]u \stackrel{\text{def}}{=} |m|(([\![M^{\alpha\to\beta}]\!]m,\ \overline{m} : \lhd u.!l : \rhd\overline{z}.[\![N^\alpha]\!]z)$.

Note that a λ-term which inhabits a certain type is mapped to a term which just inhabits the universe. Note also, however, that the free names in the encoding owns the corresponding types of a λ-term and free variables.

Now one crucial fact is that, if two simply typed λ-terms are in distinct normal forms, their encodings are never bisimilar to each other, which is easily proved by the induction of the structure of the terms. We can also show that $[\![((\lambda x^\alpha.M^\beta)^{\alpha\to\beta}N^\alpha)^\beta]\!]u \approx [\![M^\beta[N^\alpha/x^\alpha]]\!]u$ for some appropriate

[5] We note that we should *curry* a constant function (if it takes multiple arguments) before translation, since otherwise we should introduce multi-party interaction which cannot be represented as actions in our calculus. This does not let us lose any generality since the currying which respects the operational behaviour of functions is always possible. For example, "add" is split into "add$_1$" and "add$N$" in the usual way.

u. Then, using strong normalizability and determinacy [16] of the encoded terms, as well as their invariance of bisimilarity under reduction, we can prove:

PROPOSITION 3.5 $M^\alpha =_\beta N^\alpha$ *iff* $[\![M^\alpha]\!]u \approx [\![N^\alpha]\!]u$.

Finally we note that, though the embedding may be satisfactory equationally, it may not be so semantically. This is because, in the present formulation of types, a term inhabits a universe, and a universe is semantically too flat; in fact a natural computational content of our type scheme is only that $\delta \odot \overline{\delta} \rightsquigarrow \mathbf{1}$. The equational embedding nevertheless tells us that there would be room for refinement of the present type universe, which in effect enables semantically satisfactory mapping of functional types. The solution should be left to another occasion.

4 Untyped Terms and Type Inference

4.1. The untyped terms. Practically it is convenient to be able to reconstruct typed terms from untyped terms. The construction also illuminates how typable terms differentiate from untypable terms. We first define the syntax of untyped terms.

Let us be given the set of (untyped) names, again ranged over $a, b, c, \ldots$ and $x, y, z, \ldots$. We assume an irreflexive, bidirectional mapping over names, written $\overline{a}$, denoting the co-name of a name in the untyped setting. Then the set of untyped terms are given as follows. As in the explicitly typed system, we let $P, Q, R, \ldots$ range over the untyped terms. $\mathsf{c}(\tilde{x})$ is a constant in the calculus, with a distinct arity (here a numeral) for each constant symbol.

$$\begin{array}{rcl} P & ::= & a : \mathcal{V} \mid P, Q \mid |x|P \mid !P \mid \Lambda \\ \mathcal{V} & ::= & \mathsf{c}(\tilde{x}) \mid \lhd v \mid \rhd v \mid .P \mid \mathcal{V}_1; \mathcal{V}_2 \mid [\mathcal{V}_1]\&[\mathcal{V}_2] \mid \mathsf{inl}(\mathcal{V}) \mid \mathsf{inr}(\mathcal{V}) \end{array}$$

The notions of bindings, the induced syntactic constraint on branching actions, substitution, and α-conversion are defined as in 2.3 except we do not care about type compatibility of names. Among others, the naming convention in Convention 2.2 is again assumed. Starting from axioms for constants, $\rightsquigarrow$ and $\equiv$ are based on the same definition, except for $\rightsquigarrow$ we omit all type annotations. We have at most one rule for each combination of constant actions in the form:

$$\mathsf{c}(\tilde{x}) \odot \mathsf{c}(\tilde{y}) \rightsquigarrow P$$

Then reduction relation is defined as follows.

$$\text{(com)} \quad \frac{\mathcal{V}_1 \odot \mathcal{V}_2 \rightsquigarrow P}{|\tilde{w}|(\partial,\ a : \mathcal{V}_1,\ \overline{a} : \mathcal{V}_2,\ \partial') \longrightarrow |\tilde{w}|(\partial,\ P,\ \partial')} \qquad \text{(struct)} \quad \frac{P_1' \equiv P_1 \quad P_1 \xrightarrow{l} P_2 \quad P_2 \equiv P_2'}{P_1' \xrightarrow{l} P_2'}$$

Thus when two prime terms with complementary names and complementary actions meet together, reduction can take place. One essential fact in the untyped setting is that it is possible to have an incompatible pair of prime terms, even with complementary names. This we define in the following.

DEFINITION 4.1 $P \in \mathrm{ERR}$, *read* P *contains a possible run-time error, if there is* Q *such that* $P \longrightarrow\!\!\!\!\!\rightarrow Q \equiv |\tilde{w}|(\partial, a : \mathcal{V},\ \overline{a} : \mathcal{V}', \partial')$ *where there is no* R *such that* $\mathcal{V} \odot \mathcal{V}' \rightsquigarrow R$.

One example of a term with error is: $(a : \lhd v,\ \overline{a} : \rhd xy.P,\ a : 3)$. Later the notion becomes important in the context of the typing system for untyped terms.

Given typed terms, we can strip off type annotation from them, while preserving the original operational behaviour. The following "erase" function specifies this.

(i) $Erase(a^\delta) = a$, $Erase(\mathsf{c}(\tilde{x})^C) = \mathsf{c}(\tilde{x})$, $Erase(\lhd v^{!\delta}) = \lhd v$, $Erase(\rhd v^{!\delta}) = \rhd v$, and $Erase(.P^{\mathbf{1}}) = .Erase(P)$.

(ii) $Erase(\mathcal{V}_1;\mathcal{V}_2{}^{\delta_1;\delta_2}) = Erase(\mathcal{V}_1{}^{\delta_1}); Erase(\mathcal{V}_2{}^{\delta_2})$, $Erase(\mathsf{inl}(\mathcal{V})^{\delta\oplus\delta'}) = \mathsf{inl}(Erase(\mathcal{V}^{\delta}))$, $Erase(\mathsf{inr}(\mathcal{V})^{\delta\oplus\delta'}) = \mathsf{inr}(Erase(\mathcal{V}^{\delta'}))$, and $Erase(\mathcal{V}_1\&\mathcal{V}_2{}^{[\delta_1]\&[\delta_2]}) = [Erase(\mathcal{V}_1{}^{\delta_1})]\&[Erase(\mathcal{V}_2{}^{\delta_2})]$.

(iii) $Erase(a:\mathcal{V}^{\delta}) = a : Erase(\mathcal{V}^{\delta})$, $Erase((P,Q)) = (Erase(P), Erase(Q))$, $Erase(|x|P) = |x|Erase(P)$, $Erase(!P) = !Erase(P)$, and $Erase(\Lambda) = \Lambda$.

The following holds almost by definition. We assume that rules for (untyped) constant actions coincide with those in the explicitly typed system.

PROPOSITION 4.2 *Let P and Q be explicitly typed.*

(i) *If $P \longrightarrow Q$ then $Erase(P) \longrightarrow Erase(Q)$.*

(ii) *If $Erase(P) \longrightarrow R$ then there is some Q such that $P \longrightarrow Q$ and $Erase(Q) \equiv R$.*

In the present exposition we will not go into detailed study of untyped transition relation and untyped bisimilarity. Nevertheless it is worth noting that the property which holds in the case of reduction relation as stated in the above proposition does *not* hold in the case of (untyped) transition, since e.g. $b : \triangleright xy.(\overline{x} : 2,\ \overline{y} : succ(z)) \overset{b:\lfloor cc}{\longrightarrow} (c : 2,\ c : succ(z))$ where the left-hand side is typable while the right-hand side is not. Typing labels solves the issue, as discussed in [10].

4.2. Type schemes. Inferring types of an untyped term is to find the universe in which the term can safely reside. Actually it suffices to find a *portion* of a universe which is in harmony with the given term structure. No specialty arises in the formulation of the type inference system in comparison with those for functional types — in fact we only have multiple name-type pairs in *conclusion* in stead of multiple variable-type pairs in *assumption*[6]. Nevertheless the construction makes the essential idea of concurrency types (multiplicity of interfaces) explicit, and gives us additional information about the type discipline of the present system as shown in the next section.

We extend the syntax of types with type variables, ranged over by $\rho, \rho'...$ To each type variable another (different) type variable corresponds to, called *co-variable*, written $\overline{\rho}$. We set $\overline{\overline{\rho}} = \rho$ always. Then

$$\alpha ::= \rho \mid C \mid \mathbf{1} \mid \downarrow\alpha \mid \uparrow\alpha \mid \delta;\delta' \mid \delta\&\delta' \mid \delta\oplus\delta'$$

gives the set of type schemes. $a : \alpha$ (resp. $(a) : \alpha$) stands for assignment of a type to a free (resp. bound) name. The notion of co-types is defined as before. A finite set of such assignments, without collision between free names and bound names, is called a *typing*, written Γ, Δ, Θ etcetera. Intuitively a typing denotes assignment of types to names occurring in a term. $\mathcal{N}(\Gamma)$ denotes the set of names occurring in Γ. Several definitions concerning typings follow.

(i) A typing Γ is *consistent*, written $\Gamma \in \mathsf{Con}$, if names in Γ does not denote incompatible operations. That is, $\Gamma \in \mathsf{Con}$ iff $a : \alpha \wedge a : \beta$ implies $\alpha = \beta$ and $a : \alpha \wedge \overline{a} : \beta$ implies $\alpha = \overline{\beta}$, similarly for bound names.

(ii) Relatedly two typings are *compatible*, written $\Gamma \asymp \Delta$, if a common name in two predicates does not denote incompatible operations, i.e. $a : \alpha \in \Gamma$ and $b : \beta \in \Delta$ implies $\{a : \alpha,\ b : \beta\} \in \mathsf{Con}$ where a and b are the same or dual, and similarly for bound names.

(iii) Finally, since $a : \alpha$ and $\overline{a} : \overline{\alpha}$ are essentially equivalent, we define the notion of ***normal forms*** of typings. We write $|\Gamma|$ for a normal form of Γ, defined by: $\{[a] : \overline{\alpha} \mid [a] : \alpha \in \Gamma\} \cup \Gamma$ where $[a]$ may be a or (a).

4.3. Typing rules. Now we are ready to define the type inference system. It deals with two kinds of sequents. The *main sequent* is in the form $\vdash P \succ \Gamma$, which reads: the statement $P \succ \Gamma$ is derivable. The *subject* and the *predicate* of the sequent are respectively P and Γ. Then the ***auxiliary sequent*** is in the form $\Gamma \vdash \mathcal{V} : \alpha$. which reads: an auxiliary statement $\mathcal{V} : \alpha$ is derivable under the assumption Γ. The *subject*, the *predicate*, and the *assumption* of the sequent are $\mathcal{V}$, α and Γ,

[6]The sequent form was presented initially by the author in MFPS 1992, Oxford, England.

respectively. Now we give rules for typing terms. Rule (A) are for typing of terms. Rule (B) are for typing of actions. $\Gamma(\tilde{x})$ denotes names in $(\tilde{x})$ occur bound. Relatedly we have a notation $\Gamma/\tilde{x}$, which denotes the result of taking away prime statements with subjects or their conames in $\tilde{x}$, from Γ; they are used to eliminate names which occur bound from type expressions. Note, in the rules below, we are working under Convention 2.2 [7].

RULE (A)

$$\text{(prime)}\ \frac{\Gamma(\tilde{x}) \vdash \mathcal{V} : \alpha}{\vdash a : \mathcal{V} \succ a : \alpha,\ \Gamma/\tilde{x}}\ (\{a : \alpha\} \asymp \Gamma/\tilde{x}) \qquad \text{(parallel)}\ \frac{\vdash P_1 \succ \Gamma \quad \vdash P_2 \succ \Delta}{\vdash P_1,\ P_2 \succ \Gamma,\ \Delta}\ (\Gamma \asymp \Delta)$$

$$\text{(scope)}\ \frac{\vdash P \succ \Gamma}{\vdash |x|P \succ \Gamma/x} \qquad \text{(rep)}\ \frac{\vdash P \succ \Gamma}{\vdash !P \succ \Gamma} \qquad \text{(nil)}\ \frac{-}{\vdash \Lambda \succ} \qquad \text{(weak)}\ \frac{\vdash P \succ \Gamma \quad \Gamma \subset \Delta \in \mathsf{Con}}{\vdash P \succ \Delta}$$

RULE (B)

$$(\mathcal{V}\uparrow)\ \frac{-}{x : \alpha \vdash \lhd x : \uparrow\alpha} \qquad (\mathcal{V}\downarrow)\ \frac{-}{(x) : \alpha \vdash \rhd x : \downarrow\alpha} \qquad (\mathcal{V}1)\ \frac{\vdash P \succ \Gamma}{\Gamma \vdash .P : \mathbf{1}}$$

$$(\mathcal{V};)\ \frac{\Gamma \vdash \mathcal{V}_1 : \alpha_1 \quad \Delta \vdash \mathcal{V}_2 : \alpha_2}{\Gamma,\ \Delta \vdash \mathcal{V}_1;\mathcal{V}_2 : \alpha_1;\alpha_2}\ (\Gamma \asymp \Delta) \qquad (\mathcal{V}\&)\ \frac{\Gamma(\tilde{x}) \vdash \mathcal{V}_1 : \alpha_1 \quad \Delta(\tilde{x}) \vdash \mathcal{V}_2 : \alpha_2}{\Gamma,\ \Delta \vdash [\mathcal{V}_1]\&[\mathcal{V}_2] : \alpha_1\&\alpha_2}$$

$$(\mathcal{V}\oplus_l)\ \frac{\Gamma \vdash \mathcal{V} : \alpha}{\Gamma \vdash \mathsf{inl}(\mathcal{V}) : \alpha \oplus \beta} \qquad (\mathcal{V}\oplus_r)\ \frac{\Gamma \vdash \mathcal{V} : \beta}{\Gamma \vdash \mathsf{inr}(\mathcal{V}) : \alpha \oplus \beta}$$

In addition, for each constant, we assume a rule of the form: $\Gamma \vdash \mathsf{c}(\tilde{x}) : C$ where $\mathcal{N}(\Gamma) = \{\tilde{x}\}$, all names in $\tilde{x}$ are distinct and Γ consistent. If a sequent $\vdash P \succ \Gamma$ can be inferred by the above rules, then the sequent is said to be *derivable*, and P is *well-typed* under a typing Γ. Examples of derivable sequents and untypable terms follow.

(i) $\vdash u : 3 \succ u : \mathsf{nat}$.

(ii) $\vdash u : \rhd xu'.x : succ(u') \succ u : \downarrow(\downarrow\mathsf{nat})\ \downarrow\overline{\mathsf{nat}}$.

(iii) Neither $(u : 2,\ u : succ(x))$ nor $a : \lhd a$ are typable.

4.4. Basic syntactic properties. As said, the basic scheme of the type inference is an extension of the type inference for functional types in many aspects (except the duality notion), and the system we formulated in the previous subsection enjoys the syntactic properties quite similar to what we find in typing systems for functional types. Two essential results in this regard are presented below.

The first one is the subject reduction (cf. Theorem 2.6), which says that reduction does not change its potential interface. We assume that all reduction rules for constant actions are *well-typed*, i.e. if we have $\mathsf{c}(\tilde{x}) \odot \mathsf{c}'(\tilde{y}) \rightsquigarrow P$, then $\Gamma \vdash \mathsf{c}(\tilde{x}) : C$, $\Delta \vdash \mathsf{c}'(\tilde{y}) : \overline{C}$, $\vdash P \succ \Theta$, and $\Gamma \cup \Delta \cup \Theta \in \mathsf{Con}$. Then we have:

THEOREM 4.3 (Subject reduction) $\vdash P \succ \Gamma \wedge P \longrightarrow\!\!\!\!\!\rightarrow Q \ \Rightarrow\ \vdash Q \succ \Gamma$.

The proof proceeds by induction on the type derivation, where substitution of names poses a mild difficulty. We omit the details. The theorem operationally assures invariance of interface, and is tightly coupled with the lack of run time error[8]. On the latter point, we have the following, which is immediate from subject reduction.

COROLLARY 4.4 *If* $\vdash P \succ \Gamma$ *then* $P \notin$ ERR.

[7] Strictly speaking, we do not need the elimination of bound names under the convention; however, if we want more liberal assumption in the binding, the distinction between free and bound names in typing becomes indispensable (this relates to the notion of assigning types to the potential interface points of the term), so that we still keep the idea.

[8] Note if we are without the (weak) rule in Rule (A), the property becomes weaker than the above since free names will be lost during the reduction. Since typing represents potential interface of terms, the difference is not essential.

Another important property is that the typing problem is essentially computed by unification, as in the usual systems for functional types. Let a *substitution* of α for ρ in Γ be the result of substituting α for all occurrences of ρ in Γ simultaneously, written $\Gamma[\alpha/\rho]$.

THEOREM 4.5 (Principal typing [6, 14]) *Let a* substitution instance of Γ *be a result of applying substitutions zero or more times to* Γ. *Suppose* P *is well-typed. Then there is* Γ *such that* $\vdash P \succ \Gamma$ *where, for any* Γ' *with* $\vdash P \succ \Gamma'$, Γ' *is a substitution instance of* Γ. *Moreover such* Γ *can be found effectively.*

The proof uses the most common type scheme between two schemes and checks each inference rule. In [21], an algorithm which efficiently computes the most general type scheme for a restricted system, which is essentially Milner's polyadic π-calculus in [18], is presented and proved to be correct with respect to its typing system. The algorithm is easily adaptable to the system in the present paper.

4.5. Relationship with explicitly typed system. We now explicate how implicitly typed terms relate to explicitly typed terms. Given a universe $\mathcal{U}$, let $|\mathcal{U}|$ be the whole set of assignments of types to names (in the form $\{a_1 : \delta_1,\ a_2 : \delta_2,\ \ldots\}$). Then, by easy induction on the structure of typed terms, we get:

PROPOSITION 4.6 *Suppose* P (*resp.* $\mathcal{V}^\delta$) *is a term* (*resp. an action*) *in* $\mathcal{U}$. *Then for some* $\Gamma \subset |\mathcal{U}|$ *we have* $\vdash Erase(P) \succ \Gamma$ (*resp.* $\Gamma \vdash Erase(\mathcal{V}) : \delta$).

For correspondence in another direction, we first establish that if a term or an action is typable under Γ which contains no type variables, there exists a corresponding typed term or action in the explicitly typed system, which is easy under our present naming convention. But then, given $\vdash P \succ \Gamma$, by suitably choosing a substitution of type variables (write it ζ), we can always have $\vdash P \succ \Gamma\zeta$ where $\Gamma\zeta$ contains no type variables. The proposition follows, which tells us, together with Proposition 4.6, that typable terms essentially coincide in both systems[9].

PROPOSITION 4.7 *Let* P *be an untyped term. Then if* $\vdash P \succ \Gamma$ *for some* Γ, *there exists a universe* $\mathcal{U}$ *and a term* P' *in* $\mathcal{U}$ *such that, for some substitution* ζ *with* $\Gamma\zeta \subset |\mathcal{U}|$, *we have* $Erase(P') \stackrel{\text{def}}{=} P$.

5 Syntactic and Behavioural Counterpart of Typability

5.1. Preliminaries. Types for interaction we introduced above form a simple hierarchy of types, reminiscent of the simple type hierarchy in functional types. How this hierarchy is reflected in a syntactic as well as behavioural properties of terms, is studied below.

We first need some auxiliary notions regarding syntax. For simplicity, definitions are given for untyped terms, but it should be understood that similar notions are also given for explicitly typed terms.

(i) *Subexpressions*, which are terms and actions occurring in an action or in a term, are defined in a standard way. The notation is $\mathrm{Sub}(P)$ or $\mathrm{Sub}(\mathcal{V})$. For example, $\mathrm{Sub}(a : \triangleright x.\Lambda) = \{a : \triangleright x.\Lambda,\ \triangleright x.\Lambda,\ \triangleright x,\ \Lambda\}$.

(ii) We say $\mathcal{V}$ and $\mathcal{V}'$ are *compatible* if their structure of interaction is syntactically the same modulo difference in name occurrences, i.e. $\triangleleft a$ and $\triangleleft b$ are compatible, $\triangleright a$ and $\triangleright b$ are compatible, $.P$ and $.Q$ are compatible, and these extend to the composed actions.

(iii) Similarly we define the dual notion, co-compatibility. $\mathcal{V}$ and $\mathcal{V}'$ are *co-compatible*, when $\mathcal{V}$ and $\mathcal{V}'$ own a complementary syntactic structure, starting from $\triangleright v$ and $\triangleleft w$. Specifically $[\mathcal{V}]\&[\mathcal{V}']$ and $\mathsf{inl}(\mathcal{V}'')$ are co-compatible iff $\mathcal{V}$ and $\mathcal{V}''$ are co-compatible, and $.P$ and $.Q$ are always co-compatible. Details naturally follow.

[9] Without the present naming convention, the theorem holds modulo α-convertibility.

(iv) x is *active* in $\lhd x$ and $\rhd x$. If x is active in $\mathcal{V}$ so it is in $\mathcal{V};\mathcal{V}'$, $[\mathcal{V}]\&[\mathcal{V}']$, $[\mathcal{V}']\&[\mathcal{V}]$, $\mathsf{inl}(\mathcal{V})$, and $\mathsf{inr}(\mathcal{V})$. Similarly, $.P$ is active in $.P$, and if $.P$ is active in $\mathcal{V}$, so it is in $\mathcal{V};\mathcal{V}'$, $\mathcal{V}';\mathcal{V}$, $[\mathcal{V}]\&[\mathcal{V}']$, $[\mathcal{V}']\&[\mathcal{V}]$, $\mathsf{inl}(\mathcal{V})$, and $\mathsf{inr}(\mathcal{V})$.

Another essential notion is co-occurrence. Two names *co-occur* in P if they are used in the same way in actions in P. We write the co-occurrence by $\smile_P$, or simply $\smile$ when P is understood from the context. Relatedly $\frown_P$ ($\frown$ when P is understood) denotes that the co-name of a name co-occur with another name. Both relations are together generated by the following rules. Let the underlying term be P.

(i) $\smile$ is equivalence relation over the set of subexpressions of P, together with names occurring in P and their co-names (the latter not necessarily occurring in P). Also let us have $a\frown b$ iff $\overline{a}\smile b$ iff $a\smile\overline{b}$.

(ii) If $a\smile b$ and both $a:\mathcal{V}$ and $b:\mathcal{V}'$ are elements of $\mathrm{Sub}(P)$, then $\mathcal{V}\smile\mathcal{V}'$. If $a\smile b$ and both $a:\mathcal{V}$ and $\overline{b}:\mathcal{V}'$ are elements of $\mathrm{Sub}(P)$, then $\mathcal{V}\frown\mathcal{V}'$.

(iii) If $\rhd v\smile\rhd w$ or $\lhd v\smile\lhd w$ or $\rhd v\frown\lhd\overline{w}$ then $v\smile w$.

(iv) If $[\mathcal{V}_1]\&[\mathcal{V}_2]\smile[\mathcal{V}_1']\&[\mathcal{V}_2']$ or $\mathcal{V}_1;\mathcal{V}_2\smile\mathcal{V}_1';\mathcal{V}_2'$ then $\mathcal{V}_1\smile\mathcal{V}_1'$ and $\mathcal{V}_2\smile\mathcal{V}_2'$. Also if $\mathsf{inl}(\mathcal{V})\smile\mathsf{inl}(\mathcal{V}')$ or $\mathsf{inr}(\mathcal{V})\smile\mathsf{inr}(\mathcal{V}')$ then $\mathcal{V}\smile\mathcal{V}'$. If $[\mathcal{V}]\&[\mathcal{V}'']\frown\mathsf{inl}(\mathcal{V}')$ or $[\mathcal{V}'']\&[\mathcal{V}]\frown\mathsf{inr}(\mathcal{V}')$ then $\mathcal{V}\frown\mathcal{V}''$.

By easy induction, proven simultaneously with the fact that $\mathcal{V}$ and $\mathcal{V}'$ have the same type if they co-occur in P, we can get the following result. We state the result for explicitly typed terms (since the statement involves restricted names), but by Propositions 4.6 and 4.7, we have the corresponding result in the untyped terms.

PROPOSITION 5.1 *If P is a term in some universe, $a^{\delta}\smile_P b^{\delta'}\ \Rightarrow\ \delta=\delta'$ and $a^{\delta}\frown_P b^{\delta'}\ \Rightarrow\ \overline{\delta}=\delta'$.*

Now the first important syntactic notion related with our typing discipline, is the "static" counterpart of the lack of run-time error, which we call *safety*.

DEFINITION 5.2 *We say P is* safe *if, whenever $a\smile b$ (*resp. *$a\frown b$) and $a:\mathcal{V}\in Sub(P)$ and $b:\mathcal{V}'\in Sub(P)$, then $\mathcal{V}$ and $\mathcal{V}'$ are compatible (*resp. co-compatible*).*

Safety structurally ensures the operational compatibility of names which would possibly interact together. Then by Proposition 5.1 we can easily infer:

COROLLARY 5.3 *If P is a term in a certain universe, then it is safe.*

Safety itself, however, does not completely differentiate typable terms from untypable terms. As an example, take simply $a:\lhd a$. This is safe since a is equipped with only one action. But clearly the term is untypable.

5.2. Simplicity. Actually there is a clean characterization of typability based on name reference relation in a term (cf. Abramsky [1]). The concerned name reference relation is defined as follows.

DEFINITION 5.4 *We say a name a* carries *a name b in P, written $a\succ_P b$, or simply $a\succ b$ if P is understood from the context, when the following conditions hold.*

(i) $a:\mathcal{V}\in\mathrm{Sub}(P)$ and b is active in $\mathcal{V}$.

(ii) $a'\succ b$ and, moreover, $a\smile a'$ or $a\frown a'$. Or, $a\succ b'$ and, moreover, $b\smile b'$ or $b\frown b'$.

Let $\succ^+$ stands for the transitive closure of $\succ$. Given the notion, we define the essential syntactic notion corresponding to the simple type discipline.

DEFINITION 5.5 *We say a term P is* simple *if it is safe, and, moreover, for no name occurring in P we have $a \succ^+ a$.*

The simplicity denotes the lack of self-reference on names, considering co-occurrences and hereditary references. A clean characterization of the order $\succ^+$ follows.

LEMMA 5.6 *In the explicitly typed system, if $a^\delta \succ^+ b^{\delta'}$ in P, then δ' occurs in δ as its proper subexpression.*

The proof is easy, based on induction on the structure of (typed) term formation. Now the syntactic structure of a type always forms a finite tree, so the lemma implies name reference relation can never have a circular structure, so that we obtain:

PROPOSITION 5.7 *Suppose P is a term in a certain universe $\mathcal{U}$. Then P is simple. Equivalently, if $\vdash P \succ \Gamma$ for some Γ, then P is simple.*

To prove the converse, i.e. simplicity implies typability, we need to analyse the inference procedure. For the purpose, a syntactic transformation is convenient. We define the relation $\gg$ as follows. $C[\,]$ denotes an arbitrary context, except in the second rule we assume $C[\,]$ is not null, i.e. not $[\,]$. In the first rule we assume $P \not\stackrel{\text{def}}{=} \Lambda$. The elimination of "!" in the last rule is harmless by the rule (rep) in Rules (A) in 4.3.

$$C[.P] \gg (C[.\Lambda], P), \qquad C[|x|P] \gg |x|C[P], \qquad C[!P] \gg C[P]$$

It is easy that $\gg$ does not change typability nor the resulting types. In addition we easily have, if $P \gg P'$ and not $P' \gg Q$ for any Q, then P' takes a form:

$$P' \stackrel{\text{def}}{=} |\tilde{x}|(a_1 : \mathcal{V}_1, \ ..., a_n : \mathcal{V}_n)$$

where all $\mathcal{V}_n$ does not include an action of the form ".P" with $P \not\equiv \Lambda$.

Given the construction, we now show the essential reasoning to establish the desired result. Suppose P' above is simple (hence also safe). Then we can easily arrange, without changing typability, the prime terms in P' as follows: if the sequence can be written $a_1 : \mathcal{V}_1, \ ..., a_n : \mathcal{V}_n$, then in $a_1 : \mathcal{V}_1$, no names in $\mathcal{V}_1$ occur elsewhere, and all names in $\mathcal{V}_{i+1}$ occur in the preceding terms $a_j : \mathcal{V}_j$, with $j = 1..k \leq i$, and $a^{(k+1)}, .., a^i$ co-occur with a_{i+1}; moreover if a_i and a_{i+k} co-occur above, all names $a_i..a_{i+k}$ co-occur.

Then we type the term. From the structure of a term, the only significant case is the rule "parallel". Then we prove that, by induction on i and on structure of actions, if the term $(a : \mathcal{V}_1, .., a_i : \mathcal{V}_i)$ is typable, then the term $(a : \mathcal{V}_1, .., a_{i+1} : \mathcal{V}_{i+1})$ is typable. By the side condition, we only have to check the case when a_{i+1} occur as a_k with $k < i + 1$, and we can hand-calculate, using the safety condition and order relation among names, the type scheme for a_{i+1}. The main theorem follows.

THEOREM 5.8 (Characterization of typability) $\vdash P \succ \Gamma$ *for some Γ iff P is simple.*

5. 3. Simplicity and deadlock-free property. Simplicity in name reference structure may not result in meaningful behavioural characterisation immediately (in contrast to e.g. strong normalization in typable terms of simply typed λ-calculus). However for a subset of terms with a certain regular condition, the simplicity ensures one important property in concurrent computing, e.g. *deadlock-free property.* Construction below is inspired by Lafont's type discipline in [13].

We naturally work in the untyped calculus. The restricted set of terms is characterised by the regular way of generating new terms in relationship with communicating names.

DEFINITION 5.9 *A term P is* regular *if, whenever $a : \mathcal{V} \in Sub(P)$ and $.Q$ is active in $\mathcal{V}$, then we can write $Q \equiv (b_1 : \mathcal{V}_1, .., b_n : \mathcal{V}_n)$ where, for each b_i, either itself or its co-name occurs active in $\mathcal{V}$.*

Thus newly generated terms are in some way related with communication structure of the prime term. As a consequence, we have:

PROPOSITION 5.10 *Suppose $a : \mathcal{V}$ is regular. Then, if b or $\overline{b}$ occurs in $\mathcal{V}$, we have $a \succ b$.*

The proof is easy by definition of regularity. Note this implies if a or $\overline{a}$ occurs in $\mathcal{V}$ with $a : \mathcal{V}$ being regular, then P is not simple.

Now a regular term P is under *uniqueness constraint of names* if each name in P occur exactly once except by binding occurrences of scope restriction. Given the notion, a regular term is said to be *name complete* if, whenever $P \longrightarrow\!\!\!\!\!\rightarrow P'$, there is some $P'' \equiv P'$ such that (1) P'' is under uniqueness constraint, and (2) for each names occurring in P'', there exists its co-name also occurring in P''[10]. In other words, if P is name complete, all names are compensated by their co-names in any $\longrightarrow\!\!\!\!\!\rightarrow$-derivative of P[11]. But if P is name-complete, it should be possible that the term always reduces by itself; if this does not happen when some prime terms are ready to interact, this may be regarded as *deadlock*.

DEFINITION 5.11 *Suppose P is name complete. Then if $P \longrightarrow\!\!\!\!\!\rightarrow P'$ and not $P' \longrightarrow$, but P' contains a prime term as its subterm, we say P is in* deadlock.

The simplest example of deadlock is $a : \langle\widetilde{a}\rangle$. One will see that examples in deadlock all involve some kind of circular name references. Can this be proved?

To show that a name complete and typable term never has a deadlock, we first note that a name complete term only reduces to a name complete term. Thus we only have to show that if P is typable, name complete and not structurally equal to Λ or similar terms (i.e. those which contain no prime term), then it is always the case that $P \longrightarrow$.

But suppose not. Then we can write, by assumption,

$$P \equiv |\tilde{w}|(a_1 : \mathcal{V}_1, ..., a_n : \mathcal{V}_n,\ !Q_1, ..., !Q_m) \not\longrightarrow$$

where, in the right hand side, all names occur exactly once and are compensated with each other, and *either* $n \geq 1$ *or else* $m \geq 1$ with some Q_i containing a prime term. Then it is easy to eliminate replications from P without changing name reference structure and occurrences, to gain:

$$P' \stackrel{\text{def}}{=} |\tilde{w'}|(a_1 : \mathcal{V}_1, ..., a'_n : \mathcal{V}'_n) \not\longrightarrow$$

But by Proposition 5.11, the co-name of, say, a_1 cannot occur in $\mathcal{V}_1$ nor as a subject of another prime term which is not a subexpression of an action, hence it is in, say, $\mathcal{V}_2$. But then we have, by Proposition 5.11 again, $a_2 \succ a_1$. Hence the co-name of a_2 cannot occur in $\mathcal{V}_1$. Then it occurs somewhere in actions. In this way, we know that there is no place for $\overline{a'_n}$ to occur in. But by name completeness this is contradiction. Hence:

THEOREM 5.12 *Suppose for some name complete P, we have $\vdash P \succ \Gamma$. Then P never deadlocks.*

The result can be extended to terms which can be made name complete by composing another term. The characterisation is only for a restricted set of terms, but the reasoning above shows the import of name reference structure when uniqueness of names is assumed. Moreover, as suggested at the beginning, the restricted set of terms (without typability assumption) provides a term representation of Lafont's interaction net, though the translation is somewhat cumbersome[12]. We do not know, however, whether the result can generalise into less restricted set of terms, nor if significant behavioural properties other than deadlock can be associated with the present type discipline or not.

[10] We need $\equiv$ to deal with replicated terms. Note also that we are working under Convention 2.2.

[11] We note that untyped name complete terms can encode the whole scheme of partial recursive functions up to weak bisimilarity so the restricted set of terms still has non-trivial computing power.

[12] A direct way of typing Lafont's net exists and a variation of our type system can type them.

6 Discussions

6.1. Related work. The notion of typing for names in the name passing framework was already studied by Milner as *sorting* [18], and subsequently several related work including [5, 20] appeared. Our work can be seen as a further development of Milner's idea in the sense that operational structure associated with names and name passing is one of essential elements of the notion of types. The main difference lies in that the sorting discipline is not equipped with type constructors as such, basically because it only tries to capture operational structure of multiple name passing. The departure of our construction from the plain structure of sorting lies in identification of three basic types, *input*, *output*, and *term generation*, and the way of composing these types by arbitrary type constructors, among which we presented sequentialization and branching as two prominent constructs. Another, but related, point is the introduction of *constants*. The construct was naturally born when we identified the syntactic domain which represent compositional interaction patterns. But the notion is essential to form a simple hierarchy of types in the explicit setting and elegantly transplants the functionality of "constant functions" in the interaction setting, as we saw in Section 3. As constant functions are essential in pragmatic sequential languages, they would also get essential in the concurrent programming languages based on interaction. Such further significance of the construct will be treated in our coming exposition.

Other related work includes Abramsky's process interpretation of Linear Logic [1], from which we got essential suggestions regarding compositional type structure for interaction and its materialization as terms, Lafont's construction in [13], which treats a type discipline as discussed in Section 5. Our work differs from theirs in formulating a typed formalism in the general framework of process calculi (e.g. Church-Rosser is not assumed), and integration of various elements which we believe to be essential for typed concurrency into a single framework, such as abstract type constructors for interaction, constants, typed behavioural equivalences, and etcetera.

6.2. Further issues. We have only started the study of type notion in concurrent computation, constructing what may be compared to a simple hierarchy of types in the functional setting. Many remaining issues arise naturally. First, in relationship with functional types from which we got important suggestions at many stages of formal development, a systematic incorporation of various type disciplines developed in the context of functional types (cf. e.g. [2, 19]) into the present framework, is one of important subjects to study. Specifically, clarifications of behavioural consequences of such incorporation in the line of our result in 5.3 should be pursued. In this regard the work by Pierce and Sangiorgi [20] incorporates subtyping notion into Milner's sorting discipline based on refinement of usage of names in communication, and shows an interesting step in this direction. We also note that we have not touched any computability issue in the present paper. We suspect that it would be impossible to have a satisfactory encoding of general recursive functions under the simple type discipline we developed in the present paper[13]; the verification (or negation) of the conjecture is left to further study.

Another theme is enrichment of the type constructors themselves, adding such construct as parallel composition and nondeterministic branching. Some ramification of type structures would be needed to make the constructors well-suited for realistic concurrent programming, but we hope the present simple scheme would provide the basis for such development.

In the final remark of Section 3, we noted that, while we may have semantics of universe, we may not have semantics of types in the present setting. This is (essentially) due to the nature of *parallel composition*, which is not as "tight" as e.g. *functional composition* in composing behaviour of two terms. Note the lack of semantics of types means that types tell us very little about semantic (or computational) consequence of (well-typed) composition. A question is, thus, whether there is any possible framework of typed concurrency where we have actually significant semantic interpretation of each type. The issue is deep and we leave the question open.

Clearly more investigation, both theoretical and practical, is necessary to acquire sound understanding of types in concurrency. We wish that our present work will turn out to be useful as a basis

[13] If we allow recursively defined types the representation becomes possible.

of further study of typed concurrent computing.

Acknowledgements. Many thanks to Samson Abramsky, Rod Burstall, Robin Milner, Atsushi Ohori, Prakash Panangaden, Vasco Vasconcelos and Nobuko Yoshida, for enlightenment, criticism, advice, and suggestions. I also thank anonymous referees who provided valuable comments and pointed out several errors. My deep gratitude to Mario Tokoro for his long-lasting encouragement. A generous support from JSPS Fellowships for Japanese Junior Scientists is gratefully acknowledged.

References

[1] Abramsky, S., Computational interpretations of linear logic. Technical Report DOC 90/20, Imperial College, Department of Computing, October 1990. to appear in Theoretical Computer Science.

[2] Barendregt, H. and Hemerik, K., Types in lambda calculi and programming languages. In *Proceeding of ESOP 90*, 1990.

[3] Boudol, G., Asynchrony and π-calculus. Manuscript. 1992.

[4] Engberg, U. and Nielsen, M., *A Calculus of Communicating Systems with Label Passing.* Research Report DAIMI PB-208, Computer Science Department, University of Aarhus, 1986.

[5] Gay, S. J., A sort inference algorizm for the poliadic π-calculus. *POPL*, 1993.

[6] Hindley R., *The Principal Type-Scheme in Objects in Combinatory Logic*, Trans. American. Math. Soc. 146.

[7] Honda, K. and Tokoro, M., An Object Calculus for Asynchronous Communication, In: *Proc. of European Conference on Object-Oriented Programming*, LNCS, Springer-Verlag, July 1991.

[8] Honda, K., *Representing Functions in an Object Calculus*, a typescript, 19pp, October 1991. Revised version as Keio CS Report 92-005, 1992.

[9] Honda, K., *Two Bisimilarities in ν-calculus*, September 1992, submitted. Revised version as Keio CS Report 92-002, 1992.

[10] Honda, K., *Types for Dyadic Interaction* (the full version), Keio CS Report 92-003, 1993.

[11] Honda, K., *Types for Dyadic Interaction*, in CONCUR '93, LNCS, Springer-Verlag, August 1993.

[12] Honda, K., and Yoshida, N., *On Reduction-Based Process Semantics*, in Foundation of Software Technology and Theoretical Computer Science, LNCS, Springer-Verlag, December 1993.

[13] Lafont, Y., Interaction Nets, *POPL* 1990.

[14] Milner, R., A Theory of Type Polymorphism in Programming, *Journal of ACM*, 1978.

[15] Milner, R., Parrow, J.G. and Walker, D.J., *A Calculus of Mobile Processes. Part I, II.* ECS-LFCS-89-85/86, Edinburgh University, 1989.

[16] Milner, R., Functions as Processes. In *Automata, Language and Programming*, LNCS 443, 1990.

[17] Milner,R. and Sangiriorgi,D., Barbed Bisimulation, *ICALP*, 1992.

[18] Milner, R., *Polyadic π-calculus*, LFCS report, Edinburgh University 1992.

[19] Mitchell, J., Type Systems for Programming Languages, *Handbook of Theoretical Computer Science*, P 367-458, Elsevier Science Publishers B.V., 1990.

[20] Pierce, B. and Sangiorgi, D., *Typing and Subtyping for Mobile Processes.*

[21] Vasco, V. and Honda, K., *Principal typing scheme for polyadic π-Calculus*, in CONCUR '93, LNCS, Springer-Verlag, August 1993.