Principal typing schemes in a polyadic π-calculus*

Vasco T. Vasconcelos Kohei Honda
vasco@mt.cs.keio.ac.jp *kohei@mt.cs.keio.ac.jp*
Department of Computer Science
Keio University
3-14-1 Hiyoshi Kohoku-ku Yokohama 223
Japan

Abstract

The present paper introduces a typing system for a version of Milner's polyadic π-calculus, and a typing inference algorithm linear on the size of the input. The central concept underlying the typing system is the notion of type assignment, where each free name in a term is assigned a type, the term itself being given multiple name-type pairs. This observation leads to a clean typing system for Milner's sorting, and induces an efficient algorithm to infer the typing of a term. The typing system enjoys a subject-reduction property and possesses a notion of principal typing scheme. The algorithm to reconstruct the principal typing scheme of a process, or to detect its inexistence, is proved correct with respect to the typing system.

Introduction

Type discipline helps programmers not only in writing type-correct programs but especially in writing them in a principled and clear way. Concurrent programming, however, has long lacked such discipline, in contrast to functional and algebraic frameworks for sequential programs.

The present paper proposes a system of implicit typing for polyadic π-terms along the lines of type systems for untyped λ-terms, together with an efficient type reconstruction algorithm which computes the most general type of a term. In particular we concentrate on a process calculus equipped with the capability to pass arbitrary sequences of names over channels, namely a straightforward generalization of the monadic π-calculus as presented in [9], where ordered tuples of names replace single names as the object of communications. The notion of types we introduce for this calculus is essentially Milner's sorting [10]. Thus our results extend Milner's work on sorting, with a clean type inference system together with an efficient type reconstruction algorithm.

*In *4th International Conference on Concurrency Theory*, volume 715 of *LNCS*, pages 524–538. Springer-Verlag, August 1993.

The typing system presented herein was developed on the basis of an early version of a type system discussed in [6]. The essential idea underlying these systems is the notion of attributing types to names, not to terms, resulting in a scenario where a term is assigned multiple name-type pairs. We leave to [6] discussions on various type structures for dyadic interactions.

The only primitive operation for forming types, builds from types $\alpha_1, \ldots \alpha_n$ the type $(\alpha_1 \cdots \alpha_n)_n$, for $n \geq 0$, representing a property of a name carrying a sequence of names of types $\alpha_1, \ldots \alpha_n$. In this setting, a term is typable, if whenever one of its names y carries a sequence of names $x_1, \ldots x_n$ belonging to types $\alpha_1, \ldots \alpha_n$ respectively, y belongs to type $(\alpha_1 \cdots \alpha_n)_n$. A well-typing for a term is a set of assignments of types to names that makes the term typable. An algorithm is presented, which verifies if a term is typable, and in this case infers the principal typing of the term (cf. [5, 3]), from which all typings that make the term typable can be extracted. A version of the algorithm is implemented in Standard ML and has been in use for over a year.

The outline of the paper is as follows. The next section introduces our version of the polyadic π-calculus. Section 2 presents types, a basic typing system and its extension with recursive types. Section 3 presents an algorithm to reconstruct the principal typing scheme of a term along with a proof of its correctness. The last section contains discussions on related work and pointers for the future.

1 A polyadic π-calculus

This section introduces a polyadic π-calculus to the extent needed for typing considerations. The particular calculus proposed is a straightforward generalization of the monadic π-calculus, as presented in [9]. Structural congruence caters for equivalence of terms over concrete syntax and, together with normal forms introduced in [7], make the formulation of the transition relation quite concise. Summation is not considered; though its incorporation in the present system is straightforward.

The syntax of polyadic π-processes is obtained from the monadic π-calculus [9], by simply replacing single values, by sequences of values as the object of communications. Parenthesis around input names were omitted.

Definition 1.1 (Syntax) Let $\mathbf{N}$ be an infinite set of *names* and $\mathbf{N}^*$ the set of (finite) sequences over $\mathbf{N}$. The set of *polyadic π-terms* $\mathbf{C}$ is given by the following grammar.

$$P ::= \mathbf{0} \mid a\tilde{v}.P \mid \overline{a}\tilde{x}.P \mid P \mid Q \mid \nu x P \mid !P$$

where $a, b, \ldots v, x, \ldots$ range over $\mathbf{N}$; $\tilde{v}, \tilde{x}, \ldots$ range over $\mathbf{N}^*$ and $P, Q, \ldots$ range over $\mathbf{C}$. Names in $\tilde{x}$ should be pairwise distinct.

We will call processes of the form $a\tilde{v}.P$ *receptors*, and processes of the form $\overline{a}\tilde{x}.P$ *emitters*. Emitters of the form $\overline{a}\tilde{v}.\mathbf{0}$ will be simply written $\overline{a}\tilde{v}$. Multiple name restrictions on a process $\nu x_1 \cdots \nu x_n P$ will be written $\nu \tilde{x} P$. If $\tilde{x}$ is a sequence of names, then $len(\tilde{x})$ denotes the number of names in $\tilde{x}$, and if $len(\tilde{x}) = n$ we call $\tilde{x}$ a n-ary communication. We will often write $\tilde{x}_n$ to denote a sequence of names of length n. By $\{\tilde{x}\}$ we will mean the set of names occurring in $\tilde{x}$.

Receptors and scope restriction are the name binding operators in the calculus, which justifies the following definition.

Definition 1.2 (Free names) The set of *free names* in P, denoted by $\mathcal{FN}(P)$, is inductively defined by: $\mathcal{FN}(\mathbf{0}) = \emptyset$, $\mathcal{FN}(\overline{a}\tilde{v}.P) = \{a\} \cup \{\tilde{v}\} \cup \mathcal{FN}(P)$, $\mathcal{FN}(a\tilde{x}.P) = \{a\} \cup (\mathcal{FN}(P) \setminus \{\tilde{x}\})$, $\mathcal{FN}(P \mid Q) = \mathcal{FN}(P) \cup \mathcal{FN}(Q)$, $\mathcal{FN}(\nu x P) = \mathcal{FN}(P) \setminus \{x\}$, $\mathcal{FN}(!P) = \mathcal{FN}(P)$.

A notion of *substitution* of free occurrences of x by z in P, denoted $P[z/x]$, is defined in the usual way, and so is α-conversion. Also, whenever $len(\tilde{z}) = len(\tilde{x})$ and the names in $\tilde{x}$ are all distinct, $P\{z/x\}$ denotes the result of the *simultaneous replacement* of free occurrences of $\tilde{x}$ by $\tilde{z}$ in P (with change of bound names where necessary, as usual.)

Definition 1.3 (Structural congruence) $\equiv$ is the smallest congruence relation over processes defined by the following rules.

i. $P \equiv Q$ whenever P α-convertible to Q

ii. $(\mathbf{C}, \mid, \mathbf{0})$ is a commutative monoid with equality $\equiv$

iii. $\nu x P \mid Q \equiv \nu x(P \mid Q)$ whenever $x \notin \mathcal{FN}(Q)$

iv. $!P \equiv P \mid !P$

There is a certain normal form into which all terms can be transformed. Let us call terms of the form $\overline{a}\tilde{v}.P$, $a\tilde{x}.P$, $!P$ and $\mathbf{0}$ *base terms*. $\partial, \partial' \ldots$ will denote concurrent composition of base terms.

Proposition 1.4 (Normal form [7]) *Any process P in* $\mathbf{C}$ *has a structural congruent process P' of the form,*

$$\nu\tilde{u}(P_1 \mid \cdots \mid P_m)$$

where $P_1, \ldots P_m$ denote base terms, for some $m \geq 1$. Process P' (usually not unique) is called a normal form *of P.*

Reduction models the computing mechanism of the calculus. By using structural congruence and normal forms, it can be concisely defined.

Definition 1.5 (Reduction)

i. *One-step reduction*, denoted by $\rightarrow$, is the smallest relation generated by the following rules.

$$\text{COM} \qquad \nu\tilde{u}(\partial \mid \overline{a}\tilde{v}.P \mid a\tilde{x}.Q \mid \partial') \rightarrow \nu\tilde{u}(\partial \mid P \mid Q\{\tilde{v}/\tilde{x}\} \mid \partial') \qquad (len(\tilde{v}) = len(\tilde{x}))$$

$$\text{STRUCT} \qquad \frac{P' \equiv P \quad P \rightarrow Q \quad Q \equiv Q'}{P' \rightarrow Q'}$$

ii. The *reduction* relation $\twoheadrightarrow$ is the reflexive and transitive closure of one-step reduction.

2 Types and typing assignment

This section introduces a basic typing system to assign types to free names in processes, and a notion of principal typing schemes. The system is then generalized in order to accommodate recursive types.

The system $\mathrm{TA}_{()}$

Definition 2.1 (Type schemes) Let V be an infinite set of *type-variables*. The set of *type-schemes* $\mathrm{T}_{()}$ is defined by the following grammar.

$$\alpha ::= \quad t \quad | \quad (\alpha_1 \cdots \alpha_n)_n$$

for all $n \geq 0$, where $t, u, \ldots$ range over V and $\alpha, \beta \ldots$ range over $\mathrm{T}_{()}$.

Type assignment formulas are expressions $x : \alpha$, for x a name in $\mathbf{N}$ and α a type-scheme in $\mathrm{T}_{()}$, where x is called the formula's *subject* and α its *predicate*. *Typings* are sets of formulas of the form $\{x_1 : \alpha_1, \ldots x_n : \alpha_n\}$, where no two formulas have the same name as subject. $\Gamma, \Delta \ldots$ will denote typings.

We say typings Γ and Δ are *compatible*, and denote by $\Gamma \asymp \Delta$, if and only if,

$$x : \alpha \in \Gamma \wedge x : \beta \in \Delta \quad \Rightarrow \quad \alpha = \beta$$

Proposition 2.2 *If* $\Gamma \asymp \Delta$ *and* $\Gamma' \subseteq \Gamma$, *then* $\Gamma' \asymp \Delta$.

The following notation simplifies the presentation of the type assignment system. Let $\tilde{x} = x_1 \cdots x_n$ be a sequence of distinct names, $\tilde{\alpha} = \alpha_1 \cdots \alpha_n$ a sequence of type-schemes and Γ a typing. Then, $\{\tilde{x} : \tilde{\alpha}\}$ denotes the typing $\{x_1 : \alpha_1, \ldots x_n : \alpha_n\}$; $\Gamma \cdot \tilde{x} : \tilde{\alpha}$ denotes the typing $\Gamma \cup \{\tilde{x} : \tilde{\alpha}\}$, provided names in $\tilde{x}$ do not occur in Γ; $\Gamma / \tilde{x}$ denotes the typing Γ with formulas with subjects in $\tilde{x}$ removed; and $\Gamma[z/x]$ denotes the typing obtained from Γ by substitution of subject x by name z.

Typing assignment statements are expressions $P \succ \Gamma$ for all processes P and typings Γ. We will write $\vdash_{()} P \succ \Gamma$ if statement $P \succ \Gamma$ is provable using the axioms and the rules of $\mathrm{TA}_{()}$ below. Whenever $\vdash_{()} P \succ \Gamma$, for some typing Γ, we say P is *typable* and call Γ a *well-typing* for P.

Definition 2.3 (Typing scheme assignment system $\mathrm{TA}_{()}$) $\mathrm{TA}_{()}$ is defined by the following rules.

$$\textsc{Nil} \quad \vdash \mathbf{0} \succ \emptyset \qquad\qquad \textsc{Rcpt} \quad \frac{\vdash P \succ \Gamma \cdot \tilde{x}_n : \tilde{\alpha_n}}{\vdash a\tilde{x}_n.P \succ \Gamma \cup \{a : (\tilde{\alpha_n})_n\}} \quad (\Gamma \asymp \{a : (\tilde{\alpha_n})_n\})$$

$$\textsc{Emt} \quad \frac{\vdash P \succ \Gamma}{\vdash \overline{a}\tilde{v}_n.P \succ \Gamma \cup \{a : (\tilde{\alpha_n})_n\}} \quad (\{\tilde{v}_n : \tilde{\alpha_n}\} \subseteq \Gamma \text{ and } \Gamma \asymp \{a : (\tilde{\alpha_n})_n\})$$

$$\textsc{Scop} \quad \frac{\vdash P \succ \Gamma}{\vdash \nu x P \succ \Gamma / x} \qquad\qquad \textsc{Comp} \quad \frac{\vdash P \succ \Gamma \qquad \vdash Q \succ \Delta}{\vdash P \mid Q \succ \Gamma \cup \Delta} \quad (\Gamma \asymp \Delta)$$

$$\textsc{Repl} \quad \frac{\vdash P \succ \Gamma}{\vdash !P \succ \Gamma} \qquad\qquad \textsc{Weak} \quad \frac{\vdash P \succ \Gamma}{\vdash P \succ \Gamma \cdot x : \alpha}$$

Whenever a process P is typable, there exists a $\mathrm{TA}_{()}$ derivation which produces a typing containing only assignments on the free names of P. If Γ is a typing, let $\Gamma\restriction P$ be the restriction of Γ to the free names in P. We shall call typings of this form *P-typings*.

Lemma 2.4 *If $\vdash_{()} P \succ \Gamma$, then all free names of P occur in Γ and $\vdash_{()} P \succ \Gamma\restriction P$.*

Lemma 2.5 (Substitution Lemma) *Let $\vdash_{()} P \succ \Gamma$ and $x \in \mathcal{FN}(P)$.*

If z does not occur in Γ or $\{x{:}\alpha, z{:}\alpha\} \subseteq \Gamma$ for some type-scheme α, then $\vdash_{()} P[z/x] \succ \Gamma[z/x]$.

The following lemma ensures that structural congruent processes have the same typings.

Lemma 2.6 *If $\vdash_{()} P \succ \Gamma$ and $P \equiv Q$, then $\vdash_{()} Q \succ \Gamma$.*

The subject reduction property ensures that the typing of a process does not change as it is reduced and is closely related with the lack of runtime errors.

Theorem 2.7 (Subject Reduction) *If $\vdash_{()} P \succ \Gamma$ and $P \twoheadrightarrow Q$, then $\vdash_{()} Q \succ \Gamma$.*

Proof: By induction on the length of reductions. If reduction ends with the STRUCT rule, the result is immediate from lemma 2.6. We concentrate on the case reduction ends with rule COM. Then we have a sequent of the form $\vdash_{()} \nu\tilde{w}(\partial \mid a\tilde{x}.P \mid \overline{a}\tilde{v}.Q \mid \partial') \succ \Delta$. We safely assume that we have a deduction for $\vdash_{()} a\tilde{x}.P \mid \overline{a}\tilde{v}.Q \succ \Gamma$, by lemma 2.6. Now we argue that the following holds.

$$\vdash_{()} a\tilde{x}.P \mid \overline{a}\tilde{v}.Q \succ \Gamma \text{ implies } \quad \vdash_{()} P\{v/x\} \mid Q \succ \Gamma$$

Once this holds, we replace the deduction for $a\tilde{x}.P \mid \overline{a}\tilde{v}.Q$ with that for $P\{v/x\} \mid Q$, which in effect leads to the deduction of the sequent $\vdash_{()} \nu\tilde{w}(\partial \mid P\{v/x\} \mid Q \mid \partial') \succ \Delta$.

So assume $\vdash_{()} a\tilde{x}.P \mid \overline{a}\tilde{v}.Q \succ \Gamma$. At some preceding stage of the derivation, we have a sequent of the form $\vdash_{()} P \succ \Gamma_1 \cdot \tilde{x}{:}\tilde{\alpha}$ and $\Gamma_1 \subseteq \Gamma$. Similarly we have $\vdash_{()} Q \succ \Gamma_2$ with $\{\tilde{v} : \tilde{\alpha}\} \subseteq \Gamma_2$ and $\Gamma_2 \subseteq \Gamma$. Note that, since $\Gamma \asymp \Gamma$, by proposition 2.2, we have $\Gamma_1 \asymp \Gamma_2$.

Now we inspect the typing for $P\{v/x\}$. Suppose that $x_i \notin \mathcal{FN}(P)$, for some x_i in $\{\tilde{x}\}$. Then, at some preceding stage of the derivation, we have a sequent of the form $\vdash_{()} P \succ \Gamma_1'$ with $\Gamma_1' \subseteq (\Gamma_1 \cdot \tilde{x} : \tilde{\alpha})/x_i$. In this way, there is a sequent $\vdash_{()} P \succ \Gamma_1''$, with $\Gamma_1'' \subseteq \Gamma_1 \cdot \tilde{x}{:}\tilde{\alpha}$, such that names in $\{\tilde{x}\}$ which do not occur free in P, do not occur in Γ_1''. Again, by property 2.2 we have $\Gamma_1'' \asymp \Delta$. Then, if $x_i{:}\alpha_i \in \Gamma_1''$, we can substitute v_i for x_i using the substitution lemma, since $\Gamma_1'' \asymp \Gamma_2$ implies that either $v_i{:}\alpha_i \in \Gamma_1''$ or v_i does not occur in Γ_1''. Therefore, noting that if x_i does not occur in Γ_1'', then the substitution for x_i has no effect, we have $\vdash_{()} P\{v/x\} \succ \Gamma_1''\{v/x\}$. Since, $\{\tilde{v}{:}\tilde{\alpha}\} \subseteq \Gamma_2$ and $\Gamma_1'' \asymp \Gamma_2$, we have $\Gamma_1''\{v/x\} \asymp \Gamma_2$, and by rule COMP $\vdash_{()} P\{v/x\} \mid Q \succ \Gamma_1''\{v/x\} \cup \Gamma_2 \subseteq \Gamma$. Then we apply rule WEAK as many times as needed to obtain $\vdash_{()} P\{v/x\} \mid Q \succ \Gamma$, thus concluding the proof. □

Notice that the converse of subject-reduction does not hold, since non typable terms can be reduced to typable ones (e.g. $\overline{a}a \mid ax \twoheadrightarrow \mathbf{0}$), and also because free-names may be

lost in the course of reduction (e.g. $\vdash_{()} \mathbf{0} \succ \emptyset$ and $\overline{a} \mid a \twoheadrightarrow \mathbf{0}$ but $\not\vdash_{()} \overline{a} \mid a \succ \emptyset$). Also due to the loss of free names during reduction, if $\vdash_{()} P \succ \Gamma$, $P \twoheadrightarrow Q$, and $\vdash_{()} Q \succ \Delta$, then we have $\Delta\restriction Q \subseteq \Gamma\restriction P$.

A consequence of the subject-reduction property is that a typable programs will not run into type errors during execution. We say P contains a possible *runtime error*, and write $P \in \textsc{Err}$, if there exists a term Q such that $P \twoheadrightarrow Q \equiv \nu\tilde{u}(\partial \mid \overline{a}\tilde{v}.Q_1 \mid a\tilde{x}.Q_2 \mid \partial')$ and $len(\tilde{v}) \neq len(\tilde{x})$.

Corollary 2.8 *If P is typable, then $P \notin \textsc{Err}$.*

Principal typing schemes

The main interest in the introduction of type-variables is that we can see a typing for a given process as a scheme that can generate new typings for the process, by instantiating variables in the typing with types. More precisely, a *substitution on types* is a mapping $s : V \to \mathrm{T}_{()}$, from type-variables to type-schemes. If α is a type-scheme and s a substitution, then $s\alpha$ is the type-scheme obtained by replacing each variable t in α with st. A substitution s *applied to a typing* Γ is the typing $s\Gamma$ defined by $s\Gamma = \{x{:}s\alpha \mid x{:}\alpha \in \Gamma\}$.

We say a typing Δ is an *instance* of Γ if there is a substitution s such that $s\Gamma \subseteq \Delta$. Every instance of a well-typing is also a well-typing.

Lemma 2.9 *Let Δ be an instance of Γ. If $\vdash_{()} P \succ \Gamma$, then $\vdash_{()} P \succ \Delta$.*

All possible typings for a given process are instances of its *principal typing scheme.*

Definition 2.10 (Principal typing scheme) A P-typing Γ_p is *principal* for a process P if and only if,

i. $\vdash_{()} P \succ \Gamma_p$ and

ii. if $\vdash_{()} P \succ \Gamma$, then Γ is an instance of Γ_p.

It is easy to see that the principal typing scheme, when it exists, is unique up to renaming of type-variables.

Theorem 2.11 (Principal typing scheme property for $\mathbf{TA}_{()}$) *If process P is typable in $TA_{()}$, then there exists a principal typing for P. Moreover, such a typing can be found in an effective way.*

An algorithm to extract the typing of a process in a more general setting is described in section 3.

The system $\mathrm{TA}_{()\mu}$

One of the main drawbacks of the system so far introduced is that there are many meaningful and useful processes that cannot be typed. Consider the case of a list built from *Cons* and *Nil* cells defined as follows.

$$\begin{aligned} Cons(lht) &\stackrel{\text{def}}{=} lcn.\overline{c}ht \\ Nil(l) &\stackrel{\text{def}}{=} lcn.\overline{n} \end{aligned}$$

In both constructors, link l carries a pair of names: c intended to reply ht (the head and the tail of the list) in case the list is a *Cons* cell, and n intended to signal back the empty list, in case the list is a *Nil* cell. In the case of the *Cons* cell, it is easy to see that no well-typing can assign to names l and t the same (finite) type, for we would need a type α such that $\alpha = ((\beta\alpha)_2\gamma)_2$, for some types β and γ.

So, we would like to have some means for defining circular type expressions. Following Cardone and Coppo [1], we can use equations between type expressions whose solutions will be *infinite* types, i.e. type expressions whose construction trees are infinite labelled trees. So, for example to a type expression α defined by the equation $\alpha = ((t\,\alpha)_2()_0)_2$ where t is some type-variable, we associate the infinite tree depicted in figure 1.

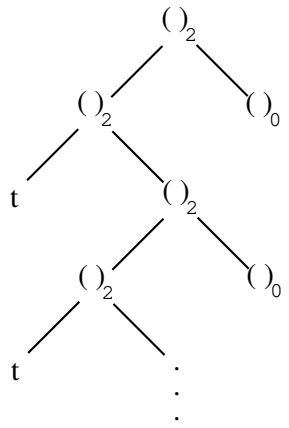

Figure 1: The infinite tree associated with equation $\alpha = ((t\,\alpha)_2()_0)_2$

Instead of using an equation for recursively specifying a type, we can introduce a special notation denoting its solution: if t is a type variable and α is an arbitrary type-scheme, then $\mu t.\alpha$ will stand for the solution of the type equation $t = \alpha$.

Definition 2.12 (Recursive type-schemes) The set of *recursive type-schemes* $\mathrm{T}_{()\mu}$ is defined by adding to the grammar of Definition 2.1, the production,

$$\alpha ::= \mu t.\alpha$$

We can identify a (possibly infinite) labelled tree with a partial function from the set of finite strings over the alphabet of positive integers (describing paths in the tree), to a

ranked alphabet $L = V \cup \{()_i \mid i \geq 0\}$, where the rank of $v \in V$ is 0 and the rank of $()_i$ is i. Let T_∞ be the set of infinite trees over L, and let $\rho, \sigma, \ldots$ range over T_∞. Introducing a new label $\perp$, we can define a partial ordering on the set of infinite trees satisfying,

i. $\perp \sqsubseteq \rho$, for all infinite trees ρ;

ii. if $\rho_1 \sqsubseteq \rho'_1, \ldots \rho_n \sqsubseteq \rho'_n$, then $(\rho_1 \cdots \rho_n)_n \sqsubseteq (\rho'_1 \cdots \rho'_n)_n$, for all $n \geq 0$.

With respect to this ordering, the set T_∞ is a c.p.o. and any infinite tree can be seen as the least upper bound of a denumerable sequence of finite approximations.

Substitution on infinite trees is a mapping $s : V \to T_\infty$. Given a substitution s and an infinite tree ρ, we obtain the infinite tree $s\rho$ as the result of the simultaneous substitution of the tree st for each occurrence of variable t in ρ. We shall write $\rho[t := \sigma]$ for the tree $s\rho$, when substitution s assigns σ to the variable t and leaves unchanged all other variables. Any such substitution s is a continuous function from T_∞ to T_∞. A translation $(\ )^*$ from recursive types to infinite trees may be defined as follows:

$$\begin{array}{rcl} t^* & = & t \quad \text{for } t \text{ a type-variable} \\ (\alpha_1 \cdots \alpha_n)^*_n & = & (\alpha_1^*, \ldots \alpha_n^*)_n \quad \text{for } n \geq 0 \\ \mu t.\alpha^* & = & \mathbf{fix}(\lambda\rho.\alpha^*[t := \rho]) \quad \text{for } \rho \text{ an infinite tree} \end{array}$$

Trees associated with recursive types are *regular*, i.e. they have finitely many subtrees. An interpretation of recursive types as infinite trees naturally induces an equivalence relation $\approx$ on $\mathrm{T}_{()\mu}$ by putting,

$$\alpha \approx \beta \quad \text{iff} \quad \alpha^* = \beta^*$$

Definition 2.13 (Recursive typing scheme assignment system $\mathbf{TA}_{()\mu}$) $\mathrm{TA}_{()\mu}$ is defined by the rules in definition 2.3 with the addition of the following rule, and changing the side condition of rule EMT to '$\{\tilde{v}_n : \tilde{\alpha}\} \subseteq \Gamma$ and $\Gamma \asymp \{a : \beta\}$ with $\beta \approx (\tilde{\alpha})_n$'.

$$(\approx) \quad \frac{\vdash P \succ \Gamma \cdot x : \alpha}{\vdash P \succ \Gamma \cdot x : \beta} \quad (\alpha \approx \beta)$$

Example 2.14 Consider the list constructors introduced in the beginning of this section. In order to deduce the typing of a list, we must handle both constructors simultaneously, and since the list and its tail must be of the same type, we shall deduce the sorting of the process $Cons(lvl) \mid Nil(l)$. A deduction is depicted below, where t is a variable and α denotes the type $\mu u.((tu)_2()_0)_2$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{(\textsc{Weak})^3\dfrac{\vdash_{()\mu} \mathbf{0} \succ \emptyset}{}}{\vdash_{()\mu} \mathbf{0} \succ v : t \cdot l : \alpha \cdot n : ()_0}(\textsc{Emt})}{\vdash_{()\mu} \overline{c}vl \succ v : t \cdot l : \alpha \cdot c : (t\alpha)_2 \cdot n : ()_0}(\approx)}{\vdash_{()\mu} \overline{c}vl \succ v : t \cdot l : ((t\alpha)_2()_0)_2 \cdot c : (t\alpha)_2 \cdot n : ()_0}(\textsc{Rcpt}) \quad \dfrac{\dfrac{\dfrac{\vdash_{()\mu} \mathbf{0} \succ \emptyset}{\vdash_{()\mu} \overline{n} \succ n : ()_0}(\textsc{Emt})}{\vdash_{()\mu} \overline{n} \succ n : ()_0 \cdot c : (t\alpha)_2}(\textsc{Weak})}{\vdash_{()\mu} lcn.\overline{n} \succ l : ((t\alpha)_2()_0)_2}(\textsc{Rcpt})}{\vdash_{()\mu} lcn.\overline{c}vl \succ v : t \cdot l : ((t\alpha)_2()_0)_2}(\textsc{Comp})$$

$$\vdash_{()\mu} lcn.\overline{c}vl \mid lcn.\overline{n} \succ v : t \cdot l : ((t\alpha)_2()_0)_2$$

Notice that α and $((t\alpha)_2()_0)_2$ represent the same infinite tree, i.e. $\alpha \approx ((t\alpha)_2()_0)_2$. The tree associated with α is depicted in figure 1.

Many syntactical properties of the system $\mathrm{TA}_{()}$ continue to hold when formulated in the extended system $\mathrm{TA}_{()\mu}$, notably the subject-reduction theorem 2.7, which implies that processes whose names can be assigned infinite types do not encounter runtime errors. Also, typable processes have a principal typing in $\mathrm{TA}_{()\mu}$, and this can be found in an effective way.

Theorem 2.15 (Principal typing scheme property for $\mathbf{TA}_{()\mu}$) *If process P is typable in $TA_{()\mu}$, then there exists a principal typing Γ_p for P. Moreover, there is an algorithm to construct Γ_p.*

As a final remark, we would like to note that an alternative approach to deal with recursive types due to MacQueen *et al.* [8] (cf. [1]) by replacing rule ($\approx$) by two rules allowing to fold and unfold recursive types, loses the property of principal typing schemes.

3 Typing reconstruction in $\mathbf{TA}_{()\mu}$

This section presents an algorithm to extract the principal recursive typing scheme of a process, together with the proof of its correctness with respect to system $\mathrm{TA}_{()\mu}$ and the analysis of its time and space complexity. We rely on the existence of a procedure to unify labelled (possibly regular infinite) trees. The algorithm is based on Wand [15], but we use a typing global to all subgoals generated, yielding an even simpler algorithm which has a space and time complexity linear on the size of the input term.

The algorithm

The algorithm builds, from a term P_0 with all bound names renamed to be distinct, a set of equations to be submitted to the unification procedure, together with a typing. Suppose the algorithm produces a set of equations E and a typing Γ_0, and use the unification algorithm on E. If s unifies E, then $s\Gamma_0$ is a well-typing for P_0. Conversely, if P_0 is typable, then all its P-typings are of the form $s(\Gamma_0 \restriction P_0)$, for s a unifier of E. We will write Γa for the type associated with name a in typing Γ.

Input: A term P_0 with all bound names renamed to be distinct.

Initialization: Set $E = \emptyset$, $G = \{P_0\}$ and Γ_0 to a typing assigning to all names in P_0 distinct type-variables.

Loop: If $G = \emptyset$, then halt and return E. Otherwise choose a goal P from G, delete it from G and add to E and G new equations and goals as specified below.

Case P is 0: Generate nothing.

Case P is $\overline{a}v_1 \cdots v_n.Q$: Generate the equation $\Gamma_0 a = (\Gamma_0 v_1 \cdots \Gamma_0 v_n)_n$ and the goal Q.

Case P is $ax_1 \cdots x_n.Q$: Generate the equation $\Gamma_0 a = (\Gamma_0 x_1 \cdots \Gamma_0 x_n)_n$ and the goal Q.

Case P is $Q \mid R$: Generate the goals Q and R.

Case P is $\nu x Q$ or $!Q$: Generate the goal Q.

To build the principal typing of a term P_0, we use the above algorithm on P_0 and then the unification algorithm on the resulting set of equations E. If E has no solutions, then P_0 is not typable. Otherwise, let s be the most general unifier of E. Then $s(\Gamma_0 \upharpoonright P_0)$ is the principal typing of P_0.

To show how the algorithm behaves, let us consider the list constructors of example 2.14. Let P_0 be the process $lcn.\overline{c}vl.\mathbf{0} \mid ldm.\overline{m}.\mathbf{0}$, and let Γ_0 be the typing $\{l : t_l, v : t_v, c : t_c, n : t_n, d : t_d, m : t_m\}$. We trace the execution of the algorithm by means of a table. The first column of each line describes the current goal set. We choose the left-most goal in the list and write in the first column in next line the new set of goals and in the second column the generated equations.

$\{lcn.\overline{c}vl.\mathbf{0} \mid ldm.\overline{m}.\mathbf{0}\}$		Initial state
$\{lcn.\overline{c}vl.\mathbf{0}, ldm.\overline{m}.\mathbf{0}\}$		Composition
$\{\overline{c}vl.\mathbf{0}, ldm.\overline{m}.\mathbf{0}\}$	$t_l = (t_c t_n)_2$	Receptor
$\{\mathbf{0}, ldm.\overline{m}.\mathbf{0}\}$	$t_c = (t_v t_l)_2$	Emitter
$\{ldm.\overline{m}.\mathbf{0}\}$		Inaction
$\{\overline{m}.\mathbf{0}\}$	$t_l = (t_d t_m)_2$	Receptor
$\{\mathbf{0}\}$	$t_m = ()_0$	Emitter
$\emptyset$		Inaction

Solving the equations generated using the unification algorithm, gives a solution assigning to type variable t_l the (infinite tree associated with the) recursive type $\mu u.((t_v u)_2()_0)_2$, and thus the principal P-typing of the process is,

$$\{l : \mu u.((t_v u)_2()_0)_2,\ v : t_v\}$$

Correctness and complexity of the algorithm

The following notation simplifies the statement of the invariant of the algorithm. Let s be a substitution on types. We say s solves an equation e, denoted $s \models e$, iff s unifies e. If E is a set of equations, we say s solves E, denoted $s \models E$, iff s unifies each equation e in E. Given a typing Γ_0, we write $s \models P$ to mean $\vdash_{()\mu} P \succ s\Gamma_0$, for some process P. If G is a set of goals, we write $s \models G$ iff $s \models P$ for each process P in G. Finally, we say s solves (E, G), denoted $s \models (E, G)$, iff $s \models E$ and $s \models G$. Now we can state the invariant of the algorithm.

$$\text{(Soundness)} \qquad \forall s.\ s \models (E, G) \ \Rightarrow\ \vdash_{()\mu} P_0 \succ s\Gamma_0$$

$$\text{(Completness)} \qquad \vdash_{()\mu} P_0 \succ \Gamma \ \Rightarrow\ \exists s.\ s \models (E, G) \wedge \Gamma \upharpoonright P_0 = s(\Gamma_0 \upharpoonright P_0)$$

At termination, when $G = \emptyset$, we have,

$$\forall s.\ s \models E \ \Rightarrow\ \vdash_{()\mu} P_0 \succ s\Gamma_0$$

$$\vdash_{()\mu} P_0 \succ \Gamma \ \Rightarrow\ \exists s.\ s \models E \wedge \Gamma \upharpoonright P_0 = s(\Gamma_0 \upharpoonright P_0)$$

so that the algorithm only produces well-typings for P_0, and if P_0 is typable, its P-well-typings are given by $s(\Gamma_0\restriction P_0)$ for some unifier s of the set of equations generated. To prove the correctness of the algorithm, we show that it terminates and that the initialization establishes and the loop step preserves the invariant.

Theorem 3.1 (Termination) *The algorithm always terminates.*

Proof: Each action generates subgoals involving terms strictly smaller than the original. □

Theorem 3.2 (Correctness) *The invariants are established by the initialization step and preserved by each case in the loop.*

Proof: (Soundness) The first part is trivial. The second follows by a simple induction on the structure of deductions.

(Completeness) For the first part take s to be the substitution that assigns α to t, whenever $x:t \in \Gamma_0\restriction P_0$ and $x:\alpha \in \Gamma\restriction P_0$. As for the second, we assume that $\vdash_{()\mu} P \succ \Gamma$. By induction hypothesis we know that,

$$\exists s.\ s \models (E,G) \wedge \Gamma\restriction P_0 = s(\Gamma_0\restriction P_0)$$

and we need to show that,

$$\exists s'.\ s' \models (E',G') \wedge \Gamma\restriction P_0 = s'(\Gamma_0\restriction P_0)$$

where (E',G') is the state after the action step. By taking $s' = s$, we only need to show that $s \models (E,G) \quad \Rightarrow \quad s \models (E',G')$. Cases $\mathbf{0}$ and $!P$ are trivial; we concentrate on the remaining ones.

Case $a\tilde{x}_n.P$. The induction hypothesis is $\vdash_{()\mu} a\tilde{x}_n.P \succ s\Gamma_0$, and we need to show that $s \models \Gamma_0 a = (\Gamma_0\tilde{x}_n)_n$ and $s \models P$. Let $s\Gamma_0 = \Gamma' \cdot \tilde{x}_n:\tilde{\alpha}_n$. By the typing rules, we have a derivation of $\vdash_{()\mu} a\tilde{x}_n.P \succ s\Gamma_0$, starting from $\vdash_{()\mu} P \succ s\Gamma_0$ and $\Gamma' \asymp \{a:(\tilde{\alpha}_n)_n\}$, by using rule RCPT, followed by n applications of rule WEAK. As for the equations, it is easy to prove that $s\Gamma_0 a = (s\Gamma_0\tilde{x}_n)_n = s(\Gamma_0\tilde{x}_n)_n$.
Case $\overline{a}\tilde{v}_n.P$. The induction hypothesis is $\vdash_{()\mu} \overline{a}\tilde{v}_n.P \succ s\Gamma_0$, and we need to show that $s \models \Gamma_0 a = (\Gamma_0\tilde{v}_n)_n$ and $s \models P$. Let $s\Gamma_0 = \Gamma' \cdot \tilde{v}_n:\tilde{\alpha}_n$. By rule EMT, we have a derivation of $\vdash_{()\mu} P \succ s\Gamma_0$ and $s\Gamma_0 \asymp \{a:(\tilde{\alpha}_n)_n\}$. As for the equations, it is easy to prove that $s\Gamma_0 a = (s\Gamma_0\tilde{v}_n)_n = s(\Gamma_0\tilde{v}_n)_n$.
Case $P \mid Q$. The induction hypothesis is $\vdash_{()\mu} P \mid Q \succ s\Gamma_0$, and we need to show that $s \models P$ and $s \models Q$. By rule COMP, we have derivations $\vdash_{()\mu} P \succ \Gamma'$ and $\vdash_{()\mu} Q \succ \Gamma''$, such that $\Gamma' \asymp \Gamma''$ and $s\Gamma_0 = \Gamma' \cup \Gamma''$. Starting from these derivations, we use the WEAK rule as many times as needed to obtain the result.
Case $\nu x P$. The induction hypothesis is $\vdash_{()\mu} \nu x P \succ s\Gamma_0$, and we need to show that $s \models P$. By the typing rules we have a derivation of $\vdash_{()\mu} \nu x P \succ s\Gamma_0$, starting from $\vdash_{()\mu} P \succ s\Gamma_0$, using rule SCOP followed by rule WEAK. □

Theorem 3.3 (Complexity) *There is an implementation of types for which the algorithm builds a typing and a set of equations in time and space linear to the textual representation of the input term.*

Proof: Typing Γ_0 can be built in a time proportional to the number of names in the input term P_0, and should take a space linear on the number of different names in P_0.

If we implement types with a directed acyclic graph, then each equation generated adds a single node to the graph and may be computed in a time proportional to the number of names involved. Also, since no copy is needed, each subgoal may be generated in constant time. Hence, the loop executes in a time linear on the size of P_0, and produces a number of equations and introduces a number of nodes in the graph no larger than the number of names in P_0. □

As a corollary, since the unification algorithm for infinite trees based on Union-Find and with types implemented with a directed acyclic graph is known to be linear (in all interesting cases) on the number of nodes in the graph, the algorithm to find the principal typing of a process is linear on the size of the textual representation of the input term.

4 Related Work and Discussions

Milner's sorting

Our notion of typing already appeared in Milner [10], under the name of "sorting". While there is a mild difference in outlook, our terms are, as easily seen, a subset of the terms of Milner's polyadic π-calculus. As to sorting, [10] introduced a "sort checking system" for the polyadic π-calculus, and it is not so difficult to prove that a term is typable in our system if and only if its analogue, under a certain sorting, can be sort-checked in Milner's system. Let us discuss this point a little.

Milner's sorting system tacitly avoids explicit infinite sorts. Instead, it assumes a collection $\mathcal{S}$ of *basic sorts*, and defines a *sorting* over $\mathcal{S}$ as a partial function $ob : \mathcal{S} \to \mathcal{S}^*$. Sorts are assigned to names, which in turn are equated via ob to labelled regular infinite trees. This formulation leads to a notion of *name-matching* of sorts, as opposed to our idea of *structural-matching* of types. As far as we concentrate on the syntax of this paper, our approach is more straightforward, while Milner's approach is more faithful to his syntactic categories as exposed in [10] (cf. [6]).

A set of basic sorts $\mathcal{S}$ and a sorting ob defined on $\{s_1, \ldots s_n\} \subseteq \mathcal{S}$ defines a *system of regular equations* [2] with equations of the form $s = (s_{i_1}, \ldots s_{i_k})_k$ for $ob(s_i) = s_{i_1} \cdots s_{i_n}$. Such a system has a unique solution $(\rho_1 \cdots \rho_n)$ in T_∞^n whose components are regular trees. It is then easy, using a standard technique, to build a typing Γ from the solution of the system. Conversely, trees in a finite set of regular trees are components of the unique solution of a single system of regular equations [2]. From such a system we can easily build $\mathcal{S}$ and ob. It is then easy to prove that we have $a : s$ and $\tilde{x}_n : ob(s)$, for some sort s, if and only if $\{a : (\tilde{\rho})_n, \tilde{x}_n : \tilde{\rho}\} \subseteq \Gamma$ for some sequence of infinite types $\tilde{\rho}$ (refer to [14] for details). Using this correspondence between sortings and typings, it is easy to prove that a located agent in standard form [10] is well-sorted if and only if the corresponding process in our calculus is typable (again, refer to [14] for details).

As an example, let us solve the system of regular equations induced by the sorting to

handle lists as in [10].

$$\begin{cases} \text{LIST} &= (\text{CONS, NIL})_2 \\ \text{CONS} &= (\text{VAL, LIST})_2 \\ \text{NIL} &= ()_0 \end{cases}$$

The system is solved equation by equation. By substituting the right-hand side of the second and third equations in the first, we obtain $\text{LIST} = ((\text{VAL LIST})_2()_0)_2$, whose solution we represent in $\text{TA}_{()\mu}$ by $\mu u.((\text{VAL } u)_2()_0)_2$. Compare it with the type-scheme of example 2.14.

Conversely, to deduce the system of regular equations whose solution contains the infinite tree $\rho = \mu u.((tu)_2()_0)_2^*$, we first identify the subtrees of ρ. They are $\{\rho, \sigma, t, ()_0\}$ where $\sigma = \mu u.(t(u()_0)_2)_2^*$ is the tree issued from node 1 of the tree in figure 1. Then, we introduce one unknown x_i, $i = 1, \ldots 4$, for each subtree of ρ, and define a system of regular equations as follows.

$$\begin{cases} x_1 &= (x_2, x_4)_2 \\ x_2 &= (x_3, x_1)_2 \\ x_3 &= t \\ x_4 &= ()_0 \end{cases}$$

In comparison, the present work extends Milner's work by recapturing the notion of sorts as *interfaces* to the outside (hence only free names are typed), formulating a typing inference system along the lines of those for functional types, and finally providing a clean typing inference algorithm which reconstructs the most general typing scheme for a given process quite efficiently. It seems that the combination of a *typing inference system* and a *corresponding typing reconstruction algorithm* for Milner's sorting has not appeared in the literature yet (see the next two paragraphs). We note that a version of our algorithm has been functioning (as an ML program) since May 1992, and the type inference system dates from April of the same year.

Gay's sorting algorithm

An algorithm to reconstruct the most general sorting of a term has already been reported by Simon Gay [4]. His inference system is a faithful adaptation of Milner's sort checking system, and the algorithm is based on this system. The work seems to be the first published report on an algorithm to compute Milner's sorting. However the orientation of the sort checking system itself is notably different from ours. As to the algorithm itself, we note that Gay's one is quadratic on the size of the input process while ours is linear.

Pierce and Sangiorgi

Benjamin Pierce and Davide Sangiorgi proposed in [12] a notion of subtyping in a version of the polyadic π-calculus that exactly matches ours. The form of types used are similar to our recursive types, except that they come annotated with input-output tags, meant to describe if a name is supposed to act as an input, an output, or an input-output channel. A subtype relation is obtained from the sub-tag relation that makes an input-output channel greater than both an input and an output channel. A collection of rules with

sequents of the form $\Gamma \vdash P : ok$ asserts that P is well-behaved under assumptions Γ, and well-sorted processes are proved not to run into type errors.

In comparison, we would like to note that their work is not aimed at providing algorithmic reconstruction of type-schemes, therefore different in motivation and result. What we tried to do in the present paper is to develop the basic construction of the type inference system and its algorithm, useful for further ramification of the idea. Different in orientation, we note that their approach shows the possibility of incorporating ideas developed in functional types into the sorting discipline, as well as how the refinement of the basic idea of the name sorting discipline, that is, "usage of names", result in semantic significance. As to the latter point, this line of research — not restricted to subtyping and annotations — seems prospective. As to the former, it suggests vast possibilities open for us to extend (or restrict) the present typing disciplines, by incorporating rich results developed in functional typing disciplines.

Remaining Issues

As was already discussed, we need to explore how various type disciplines developed in terms of functional types, can be adapted to the name sorting discipline, and how they can be of significance in the concurrency setting. For related results, see [6]. We also note that [6] shows that the simple type hierarchy has a clean characterization in terms of name reference structure. This should be true in e.g. typable terms in the polymorphic type discipline. Hopefully this line of study would lead us to behavioral characterization of typability, as has been done in functional type disciplines. We may assume that the development will have some reflection on the functional types themselves, since we will be able to reformulate the various issues of functional types in the new computational framework.

Relatedly, the present work should be extended with a typed variant of bisimilarity (or other behavioral equivalence). The work by Sangiorgi and Pierce already mentioned (as well as [6]) discusses interesting possibilities in this direction and, coupled with characterization issues we discussed above, should open important perspectives in *typed concurrent programming*.

Another related line of investigation [13] encompasses the introduction in the present system, of process-variables as present in the original presentation of π-calculus [11], accompanied by a notion of predicative polymorphism, and a ML-like let construct. Such a system would allow the declaration of a process of a polymorphic type, which could be used in a program multiple times with different types, instances of the type of the declared process. What makes such an extension appealing, besides the fact that programmers would reuse a piece of code rather that writing it twice, is that a typable program containing process declarations cannot run into type errors when executed, and that the calculus still possess a (feasible) notion of principal typings, from which all the typings that make the program typable can be derived.

Finally a pragmatic issue arises, which is to adapt the present framework to programming languages with name passing notions, especially (concurrent) object-oriented programming languages such as POOL. While these languages may be already equipped with a strong type discipline, the new type inference system and its algorithm should lead

to logical reconstruction of those typing systems, which will bring us both more efficiency in typing and reconsideration of the type notion in the languages. Typed semantics then would give the basis for optimization concerns. We wish our investigation presented herein would ultimately provide basic formal tools to deal with such pragmatic situations.

Acknowledgements. The authors wish to thank the referees who provided valuable comments. We are also greatly indebted to Prof. Mario Tokoro, for his continuous support and encouragement.

References

[1] Felice Cardone and Mario Coppo. Two Extensions of Curry's Type Inference System. In *Logic and Computer Science*, pages 19–75, Academic Press limited, 1990.

[2] Bruno Courcelle. Fundamental properties of infinite trees. *Theoretical Computer Science*, 25:95–169, 1983.

[3] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *9th ACM Symposium on Principles of Programming Languages*, pages 207–212, 1982.

[4] Simon J. Gay. A sort inference algorithm for the polyadic π-calculus. In *20th ACM Symposium on Principles of Programming Languages*, 1993.

[5] R. Hindley. The principal type-scheme of an object in combinatoric logic. *Transactions of the American Mathematical Society*, 146:29–60, December 1969.

[6] Kohei Honda. Types for Dyadic Interaction. In this volume. Also available as Keio University Report CS-92-3.

[7] Kohei Honda and Mario Tokoro. On Asynchronous Communication Semantics. In *Object-Based Concurrent Computing*, pages 21–51, Springer-Verlag, 1992. LNCS 612.

[8] D. MacQueen, G. Plotkin, and R. Sethi. An ideal model for recursive polymorphic types. *Information and Control*, 71:91–130, 1986.

[9] Robin Milner. Functions as Processes. *Automata, Language and Programming, Springer-Verlag, LNCS 443*, 1990. Also as Rapport de Recherche No 1154, INRIA-Sophia Antipolis, February 1990.

[10] Robin Milner. *The Polyadic π-Calculus: a Tutorial.* ECS-LFCS 91-180, University of Edinburgh, October 1991.

[11] Robin Milner, Joachim Parrow, and David Walker. *A Calculus of Mobile Processes, Part I and II.* ECS-LFCS 89-85/86, University of Edinburgh, June 1989.

[12] Benjamin Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. In *1993 IEEE Symposium on Logic in Computer Science*, June 1993.

[13] Vasco T. Vasconcelos. A predicative polymorphic type system for the polyadic π-calculus. 1993. Keio University.

[14] Vasco T. Vasconcelos and Kohei Honda. *Principal typing-schemes in a polyadic π-calculus.* CS 92-4, Keio University, 1992.

[15] Mitchell Wand. A simple algorithm and proof for type inference. *Fundamenta Informaticae*, X:115–122, 1987. North-Holland.