Principal Typing-Schemes in a Polyadic π-Calculus

Vasco Thudichum Vasconcelos
Kohei Honda

November 1992

Department of Computer Science
Graduate School of Science and Technology
Keio University

3–14–1 Hiyoshi, Kohoku-ku, Yokohama 223 Japan

For information on this technical report series,
send e-mail to "yama@yy.cs.keio.ac.jp".

Abstract

The present report introduces a typing system for a version of Milner's polyadic π-calculus, and a typing inference algorithm. 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. Furthermore, 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.

1.

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 in sequential programs.

The present report proposes a system of implicit typing for untyped polyadic π-terms along the lines of type systems for untyped λ-terms [7, 17, 3]. 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 [13], where ordered tuples of names replace single names as the object of communications.

The typing system here presented was developed based on that of $N_{\rightarrow}$ [8]. The essential idea underlying $N_{\rightarrow}$ (and that of the system here presented) is the notion of attributing types to names, not to terms, resulting in a scenario where a term is assigned multiple name-type scheme pairs. We leave to [8] discussions on more general type structures for dyadic interactions, among which Milner's sorting [14, 15] can be regarded as a particular instance.

The only primitive operation for forming types, builds from types $\alpha_1, \ldots \alpha_n$ the type $(\alpha_1 \cdots \alpha_n)_n$, $n \geq 0$, representing a property of a name carrying a sequence of names of types $\alpha_1, \ldots \alpha_n$. In this setting, a term is well-typed, 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, then 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 well-typed. An algorithm is presented, which verifies if a term is well-typed, and in this case infers the principal typing of the term (cf. [5]), from which all typings that make the term well-typed can be extracted. A version of the algorithm is implemented in Standard ML and has been in use for almost a year.

The outline of the report is as follows. The next section introduces our version of the polyadic π-calculus. Section 3 presents types, a basic typing system and its extension with recursive types. Section 4 presents the algorithm to infer the principal typing scheme of a term (cf. [6, 5]) along with a proof of its correctness. Section 5 compares our typing system to Milner's sorting program.

2.

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 [13]. Structural congruence on terms caters for equivalence of terms over concrete syntax and, together with normal forms of terms introduced in [9], 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 [13], by simply replacing scalar communications by vector communications. Parenthesis around input names were omitted.

Definition 2.0.1 (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-processes* $\mathbf{C}$ is given by the following grammar.

$$P ::= \quad \mathbf{0} \quad | \quad x\tilde{y}.P \quad | \quad \overline{x}\tilde{y}.P \quad | \quad P \mid Q \quad | \quad \nu x P \quad | \quad !P$$

where $a, b, \ldots$ and $v, x, y, \ldots$ range over $\mathbf{N}$; $\tilde{v}, \tilde{x}, \tilde{y}, \ldots$ range over $\mathbf{N}^*$ and $P, Q, \ldots$ range over $\mathbf{C}$.

We will call *receptors* to process of the form $x\tilde{y}.P$, and *emitters* to processes of the form $\overline{x}\tilde{y}.P$. 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 $\nu \tilde{x} P$. $len(\tilde{x})$ denotes the length of the sequence of names $\tilde{x}$, and if $len(\tilde{x}) = n$ we call $\tilde{x}$ a n-ary communication. $\{\tilde{x}\}$ denotes the set of names occurring in $\tilde{x}$. There are two kinds of bindings in the calculus.

i. In $a\tilde{x}.P$, names in $\tilde{x}$ occurring bound themselves, bind free occurrences of the same names in P.

ii. In $\nu x P$, x occurring bound itself, binds free occurrences of the same name in P.

Definition 2.0.2 (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 y in P, denoted $P[x/y]$, is defined in the usual way, and so is α-conversion. Also, whenever $len(\tilde{y}) = len(\tilde{x})$ and the names in $\tilde{x}$ are all distinct, $P\{\tilde{y}/\tilde{x}\}$ denotes the result of the *simultaneous replacement* of free occurrences of $\tilde{x}$ by $\tilde{y}$ in P (with change of bound names where necessary, as usual.)

Definition 2.0.3 (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. $P \mid Q \equiv Q \mid P$

iii. $(P \mid Q) \mid R \equiv P \mid (Q \mid R)$

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

v. $!P \equiv P \mid !P$

vi. If $P \equiv Q$, then $P \mid R \equiv Q \mid R$ and $\nu x P \equiv \nu x Q$

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

Proposition 2.0.4 (Normal form [9]) *For any process P in* **C** *there is a process P' in* **C**, *$P \equiv P'$, such that, for some $m \geq 0$,*

$$P' \stackrel{\text{def}}{=} \nu\tilde{u}(P_1 \mid \cdots \mid P_m)$$

where $P_1, \ldots P_m$ denote base terms. 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 2.0.5 (Reduction)

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

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

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

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

3.

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.

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

Definition 3.1.1 (Type schemes) Let K be a set of *type-constants* and V an infinite set of *type-variables*. The set of *type schemes* $\mathrm{T}_{()}$ is defined by the following grammar.

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

where $k, k', \ldots$ range over K; $t, t', \ldots$ range over V; $\alpha, \beta, \gamma, \ldots$ range over $\mathrm{T}_{()}$ and $n \geq 0$. A *type* is a type scheme containing no variables.

Informally, an expression of the form $(\alpha_1 \cdots \alpha_n)_n$ is intended to denote some collection of names carrying a n-ary communication whose elements belong to types $\alpha_1, \ldots \alpha_n$. We will often omit the subscript in composed types, and write $(\alpha_1 \cdots \alpha_n)$ for $(\alpha_1 \cdots \alpha_n)_n$.

Type assignment formulas are expressions $x : \alpha$, for x a name in $\mathbf{N}$ and α a type scheme in $\mathrm{T}_{()}$. 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. $\Gamma, \Delta, \Theta, \ldots$ will denote typings.

Let $\tilde{x} = x_1 \cdots x_n$ be a sequence of 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 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; $\mathcal{N}(\Gamma)$ denotes the set of names occurring in Γ.

We say typings Γ and Δ are *compatible in* $\mathrm{T}_{()}$, and denote by $\Gamma \asymp \Delta$, if

$$x : \alpha \in \Gamma \text{ and } x : \beta \in \Delta \text{ implies } \alpha = \beta.$$

Proposition 3.1.2 *If* $\Gamma \asymp \Delta$ *and* $\Theta \subseteq \Gamma$, *then* $\Theta \asymp \Delta$.

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

Definition 3.1.3 (Typing scheme assignment system $\mathbf{TA}_{()}$) $\mathrm{TA}_{()}$ is defined by the following rules.

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

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

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

$$\textsc{Repl} \quad \frac{\vdash P \succ \Gamma}{\vdash !P \succ \Gamma} \qquad\qquad \textsc{Weak} \quad \frac{\vdash P \succ \Gamma}{\vdash P \succ \Gamma \cup \{x:\alpha\}} \quad (x \notin \mathcal{N}(\Gamma))$$

Whenever a process P is well-typed, 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*.

Theorem 3.1.4 *If $\vdash_{()} P \succ \Gamma$, then $\mathcal{FN}(P) \subseteq \mathcal{N}(\Gamma)$ and $\vdash_{()} P \succ \Gamma\restriction P$.*

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

Lemma 3.1.5 *If $\vdash_{()} P \succ \Gamma$ and $P \equiv Q$, then $\vdash_{()} Q \succ \Gamma$.*

The following fundamental property of the type assignment system $\mathrm{TA}_{()}$ 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 3.1.6 (Subject Reduction) *If $\vdash_{()} P \succ \Gamma$ and $P \longrightarrow\!\!\!\!\!\rightarrow Q$, then $\vdash_{()} Q \succ \Gamma$.*

Notice that the converse of subject-reduction does not hold, since non well-typed terms can be reduced to well-typed ones (e.g. $\overline{a}a \mid ax \longrightarrow\!\!\!\!\!\rightarrow \mathbf{0}$), and also because free-names may be lost in the course of reduction (e.g. $\vdash_{()} \mathbf{0} \succ \emptyset$ and $\overline{a} \mid a \longrightarrow\!\!\!\!\!\rightarrow \mathbf{0}$ but $\not\vdash_{()} \overline{a} \mid a \succ \emptyset$). Also due to the loss of free names during reduction, if $\vdash_{()} P \succ \Gamma$, $P \longrightarrow\!\!\!\!\!\rightarrow Q$, and $\vdash_{()} Q \succ \Delta$, then $\Gamma\restriction P \subseteq \Delta\restriction Q$.

A consequence of the subject-reduction property is that a well-typed 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 $P \longrightarrow\!\!\!\!\!\rightarrow Q \equiv \nu\tilde{u}(\partial \mid \overline{a}\tilde{v}.Q_1 \mid a\tilde{x}.Q_2 \mid \partial')$ and $len(\tilde{v}) \neq len(\tilde{x})$.

Corollary 3.1.7 *If P is well-typed, then $P \notin \textsc{Err}$.*

3.2 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 from type-variables to type schemes, $s : V \to \mathrm{T}_{()}$. 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

$$s\Gamma = \{x : s\alpha \mid x : \alpha \in \Gamma\}$$

Compatibility of typings is preserved by substitution.

Proposition 3.2.1 *If* $\Gamma \asymp \Delta$, *then* $s\Gamma \asymp s\Delta$, *for any substitution* s.

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

Lemma 3.2.2 *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 3.2.3 (Principal typing scheme) A P-typing Γ_p is *principal* for a process P if

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

ii. if $\vdash_{()} P \succ \Gamma$, Γ a P-typing, 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 3.2.4 (Principal typing scheme property for $\mathbf{TA}_{()}$) *If* P *is well-typed, then there exists a principal* P*-typing* Γ_p. *Moreover,* Γ_p *can be found in an effective way.*

The principal typing scheme can be derived in a syntax-direct way by observing that any deduction of a typing Γ for a process P parallels the construction tree of the process. The general idea of the algorithm follows the one used to extract the principal type scheme of a type-free λ-term [12, 5], using Robinson's unification algorithm. An algorithm to extract the typing of a process in a more general setting is described in section 4.

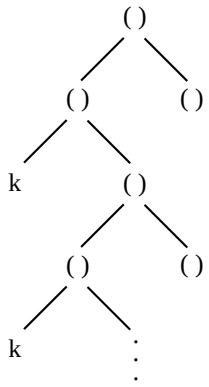

Figure 3.1 The infinite tree associated with equation $\alpha = ((k\alpha)())$

3.3 The system $TA_{()\mu}$

We would like to have some means for defining circular type expressions. For this purpose 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 $\alpha = ((k\alpha)())$ where k is some type-constant, we associate the infinite tree depicted in fig. 3.1.

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

Definition 3.3.1 (Recursive type schemes) The set of *recursive type schemes* $T_{()\mu}$ is defined by adding to the grammar of Definition 3.1.1, the production

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

We can identify a (possibly infinite) labelled tree ρ 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 $L = V \cup K \cup \{()_i \mid i \geq 0\}$, where the rank of $v \in V \cup K$ 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 infinite 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:

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

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

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

Using the equivalence relation $\approx$ on recursive types, we now say that two typings Γ and Δ are *compatible in* $\mathrm{T}_{()\mu}$, and (still) write $\Gamma \asymp \Delta$, if

$$x{:}\alpha \in \Gamma \text{ and } x{:}\beta \in \Delta \text{ implies } \alpha \approx \beta$$

Definition 3.3.2 (Recursive typing scheme assignment system $\mathbf{TA}_{()\mu}$) $\mathrm{TA}_{()\mu}$ is defined by the rules in Definition 3.1.3 with the addition of the rule

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

Example 3.3.3 A list may be defined with two constructors, Cons and Nil.

$$\begin{array}{rcl} \mathsf{Cons}(lfr) & \stackrel{\text{def}}{=} & lcn.\overline{c}fr \\ \mathsf{Nil}(l) & \stackrel{\text{def}}{=} & lcn.\overline{n} \end{array}$$

In both constructors, link l carries a pair of names: c intended to reply fr (the first and the rest 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 order to deduce the sorting of a list, we must handle both constructors simultaneously, and since the list and its rest must be of the same sort, we shall deduce the sorting of the process

$$\mathsf{Cons}(lvl) \mid \mathsf{Nil}(l)$$

A deduction is depicted below, where t is a sort-variable and α denotes the sort $\mu u.((tu)())$. Notice that α and $((\alpha t)())$ represent the same infinite tree, i.e. $\alpha \approx$

$((\alpha t)())$.

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

The tree associated with $((t\alpha)()) \approx \mu u.((tu)())$ is the one depicted in fig. 3.1, with k replaced by t.

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 3.1.6, which implies that processes whose names can be assigned infinite typings do not encounter runtime errors. Also, well-typed processes have a principal typing in $\mathrm{TA}_{()\mu}$, and this can be found in an effective way.

Theorem 3.3.4 (Principal typing scheme property for $\mathbf{TA}_{()\mu}$) *If P is a well-typed process, then there exists a principal P-typing Γ_p. Moreover, there is an algorithm to infer Γ_p.*

An algorithm to extract the principal recursive typing scheme of a process P, or to detect its inexistence, is presented and validated in the next section.

4.

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

This section presents an algorithm to extract the principal recursive typing scheme of a process. We rely on the existence of a procedure to unify labelled (possibly regular infinite) trees. According to Robinson [19] such an algorithm was hit independently by Huet [11], Baxter [2] and Robinson [18]. See [19] for an illustrated description of the algorithm.

4.1 The $\mathcal{TA}$ algorithm

Given a process P, the algorithm $\mathcal{TA}$ either succeeds producing a typing Γ such that $\vdash_{()\mu} P \succ \Gamma$ or, *fails*. When $\mathcal{TA}(P)$ succeeds, it produces a principal typing for P, and if $\mathcal{TA}(P)$ fails, there is no well-typing for P.

We assume a unification algorithm $\mathcal{U}$ accepts a set of pairs of trees to be unified and returns a substitution that makes all pairs unifiable. *Unification of typings* is defined as follows.

$$\mathit{Unify}(\Gamma, \Delta) = \mathcal{U}\{(\rho, \sigma) \mid x : \rho \in \Gamma \text{ and } x : \sigma \in \Delta\}$$

Given a sequence of names $x_1 \cdots x_n$ and a typing Γ, the *Comm* function builds a sequence of trees $\rho_1 \cdots \rho_n$ such that, for all $x_i \in \{\tilde{x}\}$, ρ_i is ρ if $x_i : \rho \in \Gamma$, or a fresh variable t_i otherwise. The $\mathcal{TA}$ function (fig. 4.1) constructs a typing on the free-names of a process by recursively descending the structure of the process. $\mathcal{TA}$ closely follows the rules of the typing assignment system, with the difference that 1) whenever, in rules EMT and RCPT, a predicate on a name is needed but not yet available, a fresh variable is created to the effect by function *Comm*, and 2) side conditions on compatibility of typings are ensured by unification.

As it happens in type inference algorithms for λ-calculus, substitutions do not need to be explicitly computed, and so a more efficient algorithm can be implemented.

4.2 Correctness of $\mathcal{TA}$

Since the use of infinite trees to deal with recursive types somewhat simplifies the treatment of syntactical properties of the typing inference system, results on this subsection are all related to infinite trees.

$$
\begin{array}{lll}
\mathcal{TA}(\mathbf{0}) & = \emptyset & \\
\mathcal{TA}(\overline{a}\tilde{v}.P) & = \textbf{let} & \Gamma = \mathcal{TA}(P) \\
 & & \tilde{\rho} = \mathit{Comm}(\tilde{v}, \Gamma) \\
 & & s = \mathit{Unify}(\Gamma \cup \{\tilde{v} : \tilde{\rho}\}, \{a : (\tilde{\rho})\}) \\
 & \textbf{in} & s(\Gamma \cup \{\tilde{v} : \tilde{\rho}\}) \cup s\{a : (\tilde{\rho})\} \\
\mathcal{TA}(a\tilde{x}.P) & = \textbf{let} & \Gamma = \mathcal{TA}(P) \\
 & & \tilde{\rho} = \mathit{Comm}(\tilde{x}, \Gamma) \\
 & & s = \mathit{Unify}(\Gamma / \tilde{x}, \{a : (\tilde{\rho})\}) \\
 & \textbf{in} & s(\Gamma / \tilde{x}) \cup s\{a : (\tilde{\rho})\} \\
\mathcal{TA}(P \mid Q) & = \textbf{let} & \Gamma = \mathcal{TA}(P) \\
 & & \Delta = \mathcal{TA}(Q) \\
 & & s = \mathit{Unify}(\Gamma, \Delta) \\
 & \textbf{in} & s\Gamma \cup s\Delta \\
\mathcal{TA}(\nu x P) & = \mathcal{TA}(P)/x & \\
\mathcal{TA}(!P) & = \mathcal{TA}(P) &
\end{array}
$$

Figure 4.1 An algorithm to extract the principal typing of a process.

A *unifier* is a substitution which makes two infinite trees syntactically equal. More precisely, a unifier of ρ and σ is a substitution $s : V \to T_\infty$ such that $s\rho = s\sigma$. A substitution s is *regular* if st is a regular infinite tree for all variables t in V. As usual, we say a unifier s of ρ and σ is *more general* then a unifier rs and say s is a *most general unifier* of ρ and σ if, for each unifier r of ρ and σ, there is a substitution u such that $r = su$. The following theorem, due to Huet [11], extends Robinson's result on unification of finite trees to infinite trees.

Theorem 4.2.1 (Unification of infinite trees [4]) *Let ρ and σ be two trees in T_∞.*

i. If ρ and σ are unifiable, they have a most general unifier $s : V \to T_\infty$. It involves variables only in ρ and in σ and is unique up to variable renaming.

ii. If ρ and σ are regular and unifiable then their most general unifier is regular. It can be effectively computed.

Let Γ and Δ be two typings with subjects in $\mathbf{N}$ and predicates in T_∞. A *unifier* of Γ and Δ is a substitution $s : V \to T_\infty$ applied to a typing, such that $s\Gamma \asymp s\Delta$. When such a substitution s exists, we say that s *unifies* Γ and Δ, and that Γ and Δ are *unifiable*. The most general unifier of two types is defined similarly to trees. It is easy to prove that if Γ and Δ are unifiable, then $\mathit{Unify}(\Gamma, \Delta)$ succeeds, and if $\mathit{Unify}(\Gamma, \Delta) = s$, then s is the most general unifier of Γ and Δ. The following theorem says that whenever $\mathcal{TA}(P)$ succeeds for some process P, then it produces a well-typing for P.

Theorem 4.2.2 (Soundness) *If $\mathcal{TA}(P)$ succeeds, then $\vdash_{()\mu} P \succ \mathcal{TA}(P)$.*

It follows by lemma 3.2.2 that every instance of $\mathcal{TA}(P)$ is a well-typing. Conversely, we can prove that every well-typing for the free-names of P is an instance of $\mathcal{TA}(P)$.

Theorem 4.2.3 (Completeness) *Suppose $\vdash_{()\mu} P \succ \Gamma$, for some typing Γ. Then,*

i. $\mathcal{TA}(P)$ succeeds;

ii. there is a substitution s such that $s\mathcal{TA}(P) = \Gamma\restriction P$.

4.3 Complexity of $\mathcal{TA}$

We now analyze the time complexity of the typing assignment algorithm.

The sequential version of the unification algorithm is known to be linear on the number of subtrees in the trees to be unified. (In fact, *Unify* executes n *union* and *find* instructions in at most $cnAck^{-1}(n)$ time, where c is a constant and Ack^{-1} is the inverse of the Ackerman function [1]. $Ack^{-1}(n) \leq 5$ for all "practical" values of n, i.e., for all $n \leq 2^{65536}$.) Since each receptor and emitter introduces a subtree in a type, it is easy to prove that the number of subtrees in a type is bound by the number of receptors and emitters in a process.

Also, it is easy to see that there will be a call to *Unify* for each pair of occurrences of the same name in a process. Finally, since both the number of occurrences of any name and the number of receptors and emitters in a process are bound by p, the textual representation of the process, we have that $\mathcal{O}(p^2)$ is an upper bound on the time taken to calculate the most general typing of a process.

5.

Comparison to Milner's sorting program

Robin Milner ([15]) introduced a different version of polyadic π-calculus along with the notion of sorts. In this section we prove that sorts are in a one-to-one correspondence with infinite regular trees; and that well-sorted (normal-form, located) agents in the Milner's polyadic π-calculus are in a one-to-one correspondence with well-typed processes in **C**.

5.1 Sorts and regular infinite trees

As it is current practice in typed functional programming languages (e.g. SML [16] and Haskell [10]), Milner's sorting program 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

$$ob : \mathcal{S} \rightarrow \mathcal{S}^*$$

Sorts are assigned to names, which in turn are equated via ob to labelled regular infinite trees. We assume ob is defined only for a finite subset of $\mathcal{S}$. Similarly to types, we write $a{:}s$ to mean that sort s is assigned to name a and, if $\tilde{x} = x_1 \cdots x_n$, $\tilde{s} = s_1 \cdots s_n$, then we write $\tilde{x}{:}\tilde{s}$ to mean $x_1{:}s_1, \ldots x_n{:}s_n$. In order to prove our first result, we make use of the theory of infinite trees [4].

Definition 5.1.1 (System of regular equations) A *system of regular equations* over a ranked alphabet L is a finite system of the form

$$U = \langle x_1 = u_1, \ldots x_n = u_n \rangle$$

where $x_1, \ldots x_n$ are the *unknowns* and $u_1, \ldots u_n$ are of the form l for l in L_0 or $l(x_{i_1}, \ldots x_{i_k})$ for l in L_k, $k \geq 1$ and $1 \geq i_1, \ldots i_k \geq n$. Let T_∞ be the set of infinite trees over L. A *solution* of U is a n-tuple of trees $(\rho_1, \ldots \rho_n)$ in T_∞^n satisfying the equations.

The second part of the following theorem is a slight generalization of a theorem in [4].

Theorem 5.1.2 (Regular systems)

i. A system of regular equations has a unique solution in T_∞; all components of this solution are regular trees.

ii. Trees in a finite set of regular trees are components of the unique solution of a single system of regular equations.

We show how to construct a typing Γ with predicates in T_∞, from a set of basic sorts $\mathcal{S}$ and a sorting ob defined on $\{s_1, \ldots s_n\} \subseteq \mathcal{S}$. First, build a system of regular equations with equations of the form $s_i = (s_{i_1}, \ldots s_{i_k})_k$, for $ob(s_i) = s_{i_1} \cdots s_{i_k}$. Such a system has a unique solution $(\rho_1, \ldots \rho_n) \in T_\infty^n$. Then, whenever $a : s$, place in Γ the assignment

$$a : \begin{cases} \rho_i & \text{if } s = s_i, 1 \leq i \leq n \\ t & \text{otherwise, } t \text{ a fresh variable} \end{cases}$$

Conversely, to build $\mathcal{S}$ and ob from Γ, first let $\langle s_1 = u_1, \ldots s_n = u_n \rangle$ be a system of equations whose solution contains the set of regular trees in Γ. Then, make $\mathcal{S} = \{s_1, \ldots s_n\} \cup K \cup V$ and

$$ob(s_i) = \begin{cases} u_i & \text{if } u_i \in K \cup V \\ s_{i_1} \cdots s_{i_k} & \text{if } u_i = (s_{i_1}, \ldots s_{i_k})_k \end{cases}$$

Correctness of the constructions is ensured by the following theorem.

Theorem 5.1.3 *Let $\mathcal{S}$,ob and Γ be related as above. Then, $a : s$ and $\tilde{x} : ob(s)$ if and only if $a : (\tilde{\rho})$ and $\tilde{x} : \tilde{\rho}$ are in Γ.*

Example 5.1.4 We solve the system of regular equations induced by the sorting to handle lists as in [15].

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

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, as usual, we represent in $\text{TA}_{()\mu}$ by $\mu t.((\text{VAL } t)())$ (subscripts omitted.) Compare it with the type scheme of Example 3.3.3.

Conversely, to deduce the system of regular equations whose solution contains the infinite tree ρ associated with the recursive type $\mu t.((kt)())$, we first identify the subtrees of ρ. They are $\{\rho, \sigma, k, ()\}$ where σ is such that $\sigma^* = \mu t.(k(t()))$ (cf. fig. 3.1). Then, we introduce one unknown x_i $(i = 1, \ldots 4)$ for each subtree of ρ, and define a system of regular equations of the form $x_i = u_i$ as follows,

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

The original system is obtained by making $x_1 = \text{LIST}$, $x_2 = \text{CONS}$ and $x_4 = \text{NIL}$.

5.2 Well-sorted located agents and well-typed processes

In [15], processes are called *agents* and are either *abstractions* or *concretions*. The idea is to factor an input (output) guarded process of the form $a(\tilde{x}).P$ ($\overline{a}\tilde{x}.P$) into a location a (co-location $\overline{a}$) and an abstraction $(\lambda\tilde{x}).P$ (concretion $[\tilde{v}].P$.) An abstraction of the form $\lambda\tilde{x}.P$ interacts with a concretion $[\tilde{v}].Q$ to produce $P \mid Q\{\tilde{v}/\tilde{x}\}$. Once located at some port, an abstraction becomes a usual input guarded process. Similarly, a concretion located at some port becomes a simple output guarded process.

Since processes in **C** do not encompass the notions of abstraction and concretion, we restrict the syntax of agents to located agents in standard form [15], by dropping from the grammar of agents the categories of abstraction and concretion. Again, we do not consider summation.

Definition 5.2.1 (Syntax of located agents) The syntax of located agents in standard form (located agents for short) **P** is given by the following grammar.

$$A ::= \quad \mathbf{0} \quad | \quad a.\lambda\tilde{x}A \quad | \quad \overline{a}.\nu\tilde{y}[\tilde{x}]A \quad | \quad A \mid B \quad | \quad \nu xA \quad | \quad !A$$

where $A, B, \ldots$ range over **P**.

Milner's program proposes a collection of formation rules ([15] pp. 34) under which an agent is said to be well-sorted, with respect to a given a sorting and an assignment of sorts to names. Bound names under different bindings are assumed to be distinct and disjoint from free names. The system below is an admissible subsystem of this collection of rules.

Lemma 5.2.2 *The following system, FR, is an admissible subsystem of the system of formation rules.*

$$\frac{a : s \quad \tilde{x} : ob(s) \quad A : ()}{a.\lambda\tilde{x}A : ()} \qquad \frac{a : s \quad \tilde{x} : ob(s) \quad A : ()}{a.\nu\tilde{y}[\tilde{x}].A : ()}$$

$$\mathbf{0} : () \qquad \frac{A : () \quad B : ()}{A \mid B : ()} \qquad \frac{A : ()}{\nu xA : ()} \qquad \frac{A : ()}{!A : ()}$$

Definition 5.2.3 (Mappings between located agents and processes)

i. $[\![\cdot]\!]_P : \mathbf{C} \to \mathbf{P}$, is defined by: $[\![\mathbf{0}]\!]_P = \mathbf{0}$, $[\![a\tilde{x}.P]\!]_P = a.\lambda\tilde{x}[\![P]\!]_P$, $[\![\overline{a}\tilde{x}.P]\!]_P = \overline{a}.[\tilde{x}][\![P]\!]_P$, $[\![P \mid Q]\!]_P = [\![P]\!]_P \mid [\![Q]\!]_P$, $[\![\nu xP]\!]_P = \nu x[\![P]\!]_P$, and $[\![!P]\!]_P = !\,[\![P]\!]_P$.

ii. $[\![\cdot]\!]_C : \mathbf{P} \to \mathbf{C}$, is defined by: $[\![\mathbf{0}]\!]_C = \mathbf{0}$, $[\![a.\lambda\tilde{x}A]\!]_C = a\tilde{x}.[\![A]\!]_C$, $[\![\overline{a}.\nu\tilde{y}[\tilde{x}]A]\!]_C = \nu\tilde{y}\overline{a}\tilde{x}.[\![A]\!]_C$ if $a \notin \{\tilde{y}\}$, $[\![A \mid B]\!]_C = [\![A]\!]_C \mid [\![B]\!]_C$, $[\![\nu xA]\!]_C = \nu x[\![A]\!]_C$, and $[\![!A]\!]_C = !\,[\![A]\!]_C$.

The following theorem states the equivalence between the notion of well-typing in $\mathrm{TA}_{()\mu}$ and that of well-sorting in Milner's system, the latter restricted to located agents.

Theorem 5.2.4

i. If A is well-sorted in FR, then $[\![A]\!]_C$ is well-typed in $TA_{()\mu}$.

ii. Conversely, if P is well-typed in $TA_{()\mu}$, then $[\![P]\!]_P$ is well-sorted in FR.

6.

Conclusion

We presented a typing system for a version of the polyadic π-calculus, where type schemes are assigned to (free) names in a process, the process itself being assigned multiple name-type scheme pairs. Well-typed processes in the system proposed were proved not to run into type errors. An efficient algorithm to compute the most general typing scheme (or to detect its inexistence) was presented and proved correct with respect to the typing system.

An interesting extension of the system encompasses the introduction of a ML-like let constructor enabling to define polymorphic processes and still keep the inference algorithm efficient.

Bibliography

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The design and analysis of computer algorithms*. Addison-Wesley Pub., 1974.

[2] L.D. Baxter. *The complexity of unification*. University of Waterloo, 1976. PhD thesis.

[3] 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.

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

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

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

[7] Roger Hindley and Jonathan Seldin. *Introduction to Combinators and λ-Calculus*. Cambridge University Press, 1986.

[8] Kohei Honda. *Types for Dyadic Interaction*. CS 92-3, Keio University, 1992.

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

[10] Paul Hudak, Joseph Fasel, and al. *Report on the Programming Language Haskell: A Non-strict, Purely Functional Language. Version 1.2*. Volume 27, ACM Sigplan Notices, May 1992. 5.

[11] G. Huet. *Résolution d'équations dans langages d'ordre* $1, 2, \ldots \omega$. These d'Etat, University Paris VII, 1976.

[12] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[13] 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.

[14] Robin Milner. Sorts and Types in the π-Calculus. December 1990. University of Edinburgh.

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

[16] Robin Milner and Mads Tofte. *The definition of Standard ML.* The MIT Press, Cambridge, Massachusetts, 1991.

[17] John C. Mitchell. *Handbook of Theoretical Computer Science*, chapter Type Systems for Programming Languages, pages 366–358. Elsevier Science Publishers B.V., 1990.

[18] J.A. Robinson. Fast Unification. In *Tagung über Automatisches Beweisen*, Mathematisches Forschungsinstitut Oberwolfach, 1976.

[19] J.A. Robinson. Logic And Logic Programming. *Communications of the ACM*, 35(3):40–65, March 1992.

Appendix A.

Proofs

3 Types and typing assignment

Proposition 3.1.2 (Typings)
Directly from the definition of typing compatibility. □

Theorem 3.1.4 (Well-typings and free names)
i $\mathcal{FN}(P) \subseteq \mathcal{N}(\Gamma)$ by a simple induction on the length of deductions.
ii $\vdash_0 P \succ \Gamma\restriction P$ by induction on the length of deductions. Cases the last rule is NIL, SCOP and REPL are trivial. Since cases EMT and RCPT present the same kind of technical difficulties, we will consider just one of them.

Case the last rule is EMT.
Suppose that $\vdash_0 P \succ \Gamma$, $\{\tilde{v} : \tilde{\alpha}\} \subseteq \Gamma$ and that $\Gamma \asymp \{a : (\tilde{\alpha})\}$. The induction hypothesis is $\vdash_0 P \succ \Gamma\restriction P$. The application of the last rule yields $\vdash_0 \overline{a}\tilde{v}.P \succ \Gamma \cup \{a : (\tilde{\alpha})\}$. Using the WEAK rule as many times as needed to introduce assignments $v_i : \alpha_i$ whenever $v_i \in \{\tilde{v}\}$ not in $\Gamma\restriction P$, we obtain $\vdash_0 P \succ \Gamma\restriction P \cup \{\tilde{v} : \tilde{\alpha}\}$. By proposition 3.1.2, $\Gamma\restriction P \cup \{\tilde{v} : \tilde{\alpha}\} \asymp \{a : (\tilde{\alpha})\}$, and by rule EMT, $\vdash_0 \overline{a}\tilde{v}.P \succ \Gamma\restriction P \cup \{\tilde{v} : \tilde{\alpha}\} \cup \{a : (\tilde{\alpha})\}$. But $\Gamma\restriction P \cup \{\tilde{v} : \tilde{\alpha}\} \cup \{a : (\tilde{\alpha})\} = (\Gamma \cup \{a : (\tilde{\alpha})\})\restriction \overline{a}\tilde{v}.P$.

Case the last rule is COMP.
Suppose that $\vdash_0 P \succ \Gamma$, $\vdash_0 Q \succ \Delta$ and that $\Gamma \asymp \Delta$. The application of the last rule yields $\vdash_0 P \mid Q \succ \Gamma \cup \Delta$. By proposition 3.1.2, $\Gamma\restriction P \asymp \Delta\restriction Q$, and by rule COMP, $\vdash_0 P \mid Q \succ \Gamma\restriction P \cup \Delta\restriction Q$. By the first part of this theorem, all the free names of P are in Γ, and similarly for Q and Δ. Hence, $\Gamma\restriction P \cup \Delta\restriction Q = (\Gamma \cup \Delta)\restriction (P \mid Q)$.

Case the last rule is WEAK.
Suppose that $\vdash_0 P \succ \Gamma$ and that x is not in Γ. The induction hypothesis is $\vdash_0 P \succ \Gamma\restriction P$. The application of the last rule yields $\vdash_0 P \succ \Gamma \cdot x : \alpha$, for some type scheme α. Since x is not in Γ, by the first part of this theorem, x is not free in P. Hence, $\Gamma\restriction P = (\Gamma \cdot x : \alpha)\restriction P$. □

The following corollary enables to consider, without loss of generality, only deductions of the form $\vdash_0 P \succ \Gamma\restriction P$.

Corollary A.0.5 *A deduction of* $\vdash_0 P \succ \Gamma$ *can be factored out in two parts:*

i. a deduction of $\vdash_0 P \succ \Gamma\restriction P$*, followed by*

ii. a series of applications of the WEAK *rule.*

Proof: Let $\vdash_{()} P \succ \Gamma\restriction P \cup \{x_1 : \alpha_1, \ldots x_n : \alpha_n\}$. By theorem 3.1.4*ii* there is a deduction $\vdash_{()} P \succ \Gamma\restriction P$. Use the WEAK rule n times to introduce type assignments $x_i{:}\alpha_i$, in the typing of P. □

The following lemma will be used in the proof of the structural congruence lemma and the subject-reduction theorem.

Lemma A.0.6 (Substitution Lemma) *Let* $\vdash_{()} P \succ \Gamma$ *and* $x \in \mathcal{FN}(P)$.

If $z \notin \mathcal{N}(\Gamma)$ *or* $\{x : \alpha, z : \alpha\} \subseteq \Gamma$, *for some type scheme* α, *then* $\vdash_{()} P[z/x] \succ \Gamma[z/x]$.

Proof: By induction on the structure of deductions. The most interesting cases are described below.

Case P is $\nu y Q$.
Suppose that $\vdash_{()} Q[z/x] \succ \Gamma[z/x]$. By rule SCOP $\vdash_{()} \nu y P[z/x] \succ \Gamma[z/x]/y$. Since x is free in $\nu y P$, $x \neq y$ and hence, $\Gamma[z/x]/y = (\Gamma/y)[z/x]$.

Case P is $a\tilde{y}.Q$.
Suppose that $\vdash_{()} Q \succ \Gamma \cdot \tilde{y}{:}\tilde{\alpha}$, $\Gamma \asymp \{a{:}(\tilde{\alpha})\}$, and that $\vdash_{()} Q[z/x] \succ (\Gamma \cdot \tilde{y}{:}\tilde{\alpha})[z/x]$. By proposition 3.2.1, $\Gamma[z/x] \asymp \{a : (\tilde{\alpha})\}[z/x]$. Since x is free in $a\tilde{y}.Q$, $x \notin \{\tilde{y}\}$, and hence, by rule RCPT $\vdash_{()} a[z/x]\tilde{y}.Q[z/x] \succ \Gamma[z/x] \cup \{a : (\tilde{\alpha})\}[z/x]$, that is $\vdash_{()} (a\tilde{y}.Q)[z/x] \succ (\Gamma \cup \{a{:}(\tilde{\alpha})\})[z/x]$. □

Lemma 3.1.5 (Structural congruence and typings) Without loss of generality, assume all deductions are of the form $\vdash_{()} P \succ \Gamma\restriction P$.
Case $P \equiv_\alpha Q$.
By induction on the structure of P. The non-trivial cases are when P is $\nu x R$ or P is $a\tilde{x}.R$.

Subcase $\nu x P \equiv_\alpha \nu z P[z/x]$, z not free in P.
Suppose that $\vdash_{()} P \succ \Gamma$. Using the SCOP rule we have $\vdash_{()} \nu x P \succ \Gamma/x$. Since z is not free in P, by assumption z is not in Γ. Also, since x is free in P, by the substitution lemma, $\vdash_{()} P[z/x] \succ \Gamma[z/x]$. Using the SCOP rule $\vdash_{()} \nu z P[z/x] \succ \Gamma[z/x]/z = \Gamma/x$.

Subcase $a\tilde{x}.P \equiv_\alpha a\tilde{x}[z/x_i].P[z/x_i]$, $x_i \in \{\tilde{x}\}$ and z not free in P.
Suppose that $\vdash_{()} P \succ \Gamma \cdot \tilde{x}{:}\tilde{\alpha}$ and that $\Gamma \asymp \{a{:}(\tilde{\alpha})\}$. Using the RCPT rule we have $\vdash_{()} a\tilde{x}.P \succ \Gamma \cup \{a{:}(\tilde{\alpha})\}$. Since z in not free in P, by assumption z is not in $\Gamma \cdot \tilde{x}{:}\tilde{\alpha}$. Also, since x_i is free in P, by the substitution lemma, $\vdash_{()} P[z/x_i] \succ \Gamma \cdot (\tilde{x}[z/x_i]{:}\tilde{\alpha})$. Using the RCPT rule $\vdash_{()} a\tilde{x}[z/x_i].P[z/x_i] \succ \Gamma \cup \{a{:}(\tilde{\alpha})\}$.

Case $P \mid Q \equiv Q \mid P$.
Follows directly from the commutativity of set union.

Case $(P_1 \mid P_2) \mid P_3 \equiv P_1 \mid (P_2 \mid P_3)$.
We have $\vdash_{()} P_i \succ \Gamma_i$, for $i = 1,2,3$, with $\Gamma_i \asymp \Gamma_j$. The result follows from the associativity of set union.

Case $\nu x P \mid Q \equiv \nu x (P \mid Q)$, $x \notin \mathcal{FN}(Q)$.
Suppose that $\vdash_{()} P \succ \Gamma$, $\vdash_{()} Q \succ \Delta$ and $\Gamma \asymp \Delta$. ($\Rightarrow$) Using the SCOP followed by the COMP rule, we have $\vdash_{()} \nu x P \mid Q \succ \Gamma/x \cup \Delta$, since $\Gamma/x \asymp \Delta$ by proposition 3.1.2.

($\Leftarrow$) Using the COMP followed by the SCOP rule, we have $\vdash_0 \nu x(P \mid Q) \succ (\Gamma \cup \Delta)/x$. Since x is not free in Q, by assumption x does not occur in Δ, and thus $(\Gamma \cup \Delta)/x = \Gamma/x \cup \Delta$.

Case $!P \equiv P \mid !P$.
Suppose $\vdash_0 P \succ \Gamma$. ($\Rightarrow$) By rule REPL, $\vdash_0 !P \succ \Gamma$. ($\Leftarrow$) Since, by proposition 3.1.2, $\Gamma \asymp \Gamma$, by rule COMP $\vdash_0 P \mid !P \succ \Gamma \cup \Gamma = \Gamma$.

Case $P \equiv Q$ implies $P \mid R \equiv Q \mid R$ and **Case** $P \equiv Q$ implies $\nu x P \equiv \nu x Q$ are trivial. □

Theorem 3.1.6 (Subject Reduction)
By induction on the lenght of deductions. If reduction ends with the STRUCT rule, the result is immediate from lemma 3.1.5. We concentrate on the case reduction ends with rule COM. Then we have a sequent of the form $\vdash_0 \nu\tilde{w}(\partial \mid a\tilde{x}.P \mid \overline{a}\tilde{v}.Q \mid \partial') \succ \Delta$. We safely assume that we have a deduction for $\vdash_0 a\tilde{x}.P \mid \overline{a}\tilde{v}.Q \succ \Gamma$, by lemma 3.1.5. Now we argue that the following holds.

$$\vdash_0 a\tilde{x}.P \mid \overline{a}\tilde{v}.Q \succ \Gamma \text{ implies } \quad \vdash_0 P\{v/x\} \mid Q \succ \Gamma$$

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 $\vdash_0 \nu\tilde{w}(\partial \mid P\{v/x\} \mid Q \mid \partial') \succ \Delta$.

So assume $\vdash_0 a\tilde{x}.P \mid \overline{a}\tilde{v}.Q \succ \Gamma$. At some preceding stage of the derivation, we have a sequent of the form $\vdash_0 P \succ \Gamma_1 \cdot \tilde{x}:\tilde{\alpha}$ and $\Gamma_1 \subseteq \Gamma$. Similarly we have, at some preceding stage of the deduction, $\vdash_0 Q \succ \Gamma_2$ with $\{\tilde{v} : \tilde{\alpha}\} \subseteq \Gamma_2$ and $\Gamma_2 \subseteq \Gamma$. Note that we also have $\Gamma_1 \asymp \Gamma_2$.

Now we inspect the typing for $P\{v/x\}$. At some preceding stage of the derivation, we have $\vdash_0 P \succ \Gamma_1'$, where $\Gamma_1' = (\Gamma_1 \cdot \tilde{x} : \tilde{\alpha})\restriction P$. By theorem 3.1.4, if $x_i \in \mathcal{N}(\Gamma_1')$, then $x_i \in \mathcal{FN}(P)$, and hence, by the substitution lemma $\vdash_0 P[v_i/x_i] \succ \Gamma_1'[v_i/x_i]$, where either $v_i \notin \mathcal{N}(\Gamma_1')$ or $v_i : \alpha_i \in \Gamma_1'$. Noting that if $x_i \notin \mathcal{N}(\Gamma_1')$, then the substitution for x_i has no effect, we get $\vdash_0 P\{v/x\} \succ \Gamma_1'\{v/x\} = (\Gamma_1 \cdot \tilde{v} : \tilde{\alpha})\restriction P$. Since $\{\tilde{v} : \tilde{\alpha}\} \subseteq \Gamma_2$, by proposition 3.1.2, $(\Gamma_1 \cdot \tilde{v} : \tilde{\alpha})\restriction P \asymp \Gamma_2$, and by rule COMP $\vdash_0 P\{v/x\} \mid Q \succ \Gamma_1\restriction P \cup \Gamma_2$. Then we apply the WEAK rule as many times as needed to obtain $\vdash_0 P\{v/x\} \mid Q \succ \Gamma$, thus concluding the proof. □

Corollary 3.1.7 (Runtime errors)
Suppose that $P \twoheadrightarrow P'$ and that P is typable. By the subject-reduction theorem, P' is typable. Let $P' \equiv \nu\tilde{w}(\partial \mid a\tilde{x}.Q_1 \mid \overline{a}\tilde{v}.Q_2 \mid \partial')$. Then, at some preceding stage of the deduction, we have sequents $\vdash_0 a\tilde{x}.Q_1 \succ \Gamma_1$ and $\vdash_0 \overline{a}\tilde{v}.Q_2 \succ \Gamma_2$ with $a \in \mathcal{N}(\Gamma_1) \cap \mathcal{N}(\Gamma_2)$ and $\Gamma_1 \asymp \Gamma_2$. This implies $len(\tilde{x}) = len(\tilde{v})$. □

Proposition 3.2.1 (Typings and substitutions)
Directly from the definition of typing compatibility. □

Lemma 3.2.2 (Well-typings and typing instances)
(Outline) By induction on the length of deductions. Since compatibility of typings is preserved by substitution, replace in the deduction of $P \succ \Gamma$, all occurrences of $x:\alpha$ by $x:s\alpha$, to obtain a deduction of $P \succ s\Gamma$. □

4 Typing inference in $\mathbf{TA}_{()\mu}$

Theorem 4.2.2 (Soundness of the typing algorithm)
A simple induction on the cases analyzed by $\mathcal{TA}$. Cases $\mathcal{TA}(\mathbf{0})$, $\mathcal{TA}(\nu x P)$ and $\mathcal{TA}(!Q)$ are trivial. Since cases $\mathcal{TA}(\overline{a}\tilde{v}.P)$ and $\mathcal{TA}(a\tilde{x}.P)$ present the same kind of technical difficulties, we will consider just one of them.

Case $\mathcal{TA}(a\tilde{x}.P)$.
Suppose that $\mathcal{TA}(P) = \Gamma$, $Comm(\tilde{x}, \Delta) = \tilde{\rho}$ and that $Unify(\Gamma/\tilde{x}, \{a : (\tilde{\rho})\}) = s$. The induction hypothesis is $\vdash_{()\mu} P \succ \Gamma$. Use the WEAK rule as many times as needed to introduce assignments $v_i : \rho_i$ ($v_i \in \{\tilde{v}\}$) whenever v_i not in Γ. We obtain $\vdash_{()} P \succ \Gamma/\tilde{x} \cup \{\tilde{v} : \tilde{\rho}\}$. By lemma 3.2.2, $\vdash_{()\mu} P \succ s(\Gamma/\tilde{x} \cup \{\tilde{v} : \tilde{\rho}\})$. Since s unifies $\Gamma/\tilde{x}$ and $\{a : (\tilde{\rho})\}$, we have $s\Gamma/\tilde{x} \asymp s\{a : (\tilde{\rho})\}$. Hence, by rule RCPT, $\vdash_{()\mu} a\tilde{x}.P \succ s\Gamma/\tilde{x} \cup s\{a : (\tilde{\rho})\}$.

Case $\mathcal{TA}(P \mid Q)$.
Suppose that $\mathcal{TA}(P) = \Gamma$, $\mathcal{TA}(Q) = \Delta$ and $Unify(\Gamma, \Delta) = s$. The induction hypothesis is $\vdash_{()\mu} P \succ \Gamma$ and $\vdash_{()\mu} Q \succ \Delta$. By lemma 3.2.2, $\vdash_{()\mu} P \succ s\Gamma$ and $\vdash_{()\mu} Q \succ s\Delta$, and by proposition 3.2.1, $s\Gamma \asymp s\Delta$. Hence, by rule COMP, $\vdash_{()} P \mid Q \succ s\Gamma \cup s\Delta$. □

Theorem 4.2.3 (Completeness of the typing algorithm)
By induction on the length of deductions. Cases the last rule is NIL, SCOP and REPL are trivial. Since cases EMT and RCPT present the same kind of technical difficulties, we will consider just one of them.

Case the last rule is EMT.
Suppose that $\vdash_{()\mu} P \succ \Delta$, $\{\tilde{v} : \tilde{\sigma}\} \subseteq \Delta$, and that $\Delta \asymp \{a : (\tilde{\sigma})\}$. The induction hypothesis are that $\mathcal{TA}(P) = \Gamma$ and that there is a substitution s' such that $s'\Gamma = \Delta\restriction P$.
i Let $Comm(\tilde{v}, \Gamma) = \tilde{\rho}$. According to the definition of $Comm$, extend s' to assignments $v_i : t_i$ ($v_i \in \{\tilde{v}\}$ and t_i a fresh variable) whenever v_i does not occur in Γ, by making $s't_i = \sigma_i$. We now have $s'\tilde{\rho} = \tilde{\sigma}$ and thus $s'(\Gamma \cup \{\tilde{v} : \tilde{\rho}\}) = (\Delta\restriction P) \cup \{\tilde{v} : \tilde{\sigma}\}$. By proposition 3.1.2, $s'(\Gamma \cup \{\tilde{v} : \tilde{\rho}\}) = (\Delta\restriction P) \cup \{\tilde{v} : \tilde{\sigma}\} \subseteq \Delta \asymp \{a : (\tilde{\sigma})\} = s'\{a : (\tilde{\rho})\}$. Hence s' unifies $\Gamma \cup \{\tilde{v} : \tilde{\rho}\}$ and $\{a : (\tilde{\rho})\}$, $Unify(\Gamma \cup \{\tilde{v} : \tilde{\rho}\}, \{a : (\tilde{\rho})\})$ succeeds and so does $\mathcal{TA}(\overline{a}\tilde{v}.P)$.
ii Since $\Gamma \cup \{\tilde{v} : \tilde{\rho}\}$ and $\{a : (\tilde{\rho})\}$ are unifiable, let $Unify(\Gamma \cup \{\tilde{v} : \tilde{\rho}\}, \{a : (\tilde{\rho})\}) = r$ where r is the m.g.u. of $\Gamma \cup \{\tilde{v} : \tilde{\rho}\}$ and $\{a : (\tilde{\rho})\}$. Since s' also unifies $\Gamma \cup \{\tilde{v} : \tilde{\rho}\}$ and $\{a : (\tilde{\rho})\}$, there is a substitution u such that $s' = ur$. Let s be u. We successively have $s\mathcal{TA}(\overline{a}\tilde{v}.P) = s(r(\Gamma \cup \{\tilde{v} : \tilde{\rho}\}) \cup r(\{a : (\tilde{\rho})\})) = s'(\Gamma \cup \{\tilde{v} : \tilde{\rho}\}) \cup s'\{a : (\tilde{\rho})\} = (\Delta\restriction P) \cup \{\tilde{v} : \tilde{\sigma}\} \cup \{a : (\tilde{\sigma})\} = (\Delta \cup \{a : (\tilde{\sigma})\})\restriction \overline{a}\tilde{v}.P$.

Case the last rule is COMP.
Suppose that $\vdash_{()\mu} P_i \succ \Delta_i$, for $i = 1, 2$, and that $\Delta_1 \asymp \Delta_2$. The induction hypothesis are that $\mathcal{TA}(P_i) = \Gamma_i$ and that there are substitutions s_i such that $s_i\Gamma_i = \Delta_i\restriction P_i$.
i By proposition 3.1.2, $s_1\Gamma_1 = \Delta_1\restriction P_1 \subseteq \Delta_1 \asymp \Delta_2 \supseteq \Delta_2\restriction P_2 = s_2\Gamma_2$. Let s' be the substitution $s_1 \cup s_2$. Then s' unifies Γ_1 and Γ_2, $Unify(\Gamma_1, \Gamma_2)$ succeeds and so does $\mathcal{TA}(P_1 \mid P_2)$.
ii Since Γ_1 and Γ_2 are unifiable, let $Unify(\Gamma_1, \Gamma_2) = r$ where r is the m.g.u. of Γ_1

and Γ_2. Since s' also unifies Γ_1 and Γ_2, there is a substitution u such that $s' = ur$. Let s be u. We successively have $s\mathcal{TA}(P_1 \mid P_2) = s(r\Gamma_1 \cup r\Gamma_2) = s'\Gamma_1 \cup s'\Gamma_2 = \Delta_1\restriction P_1 \cup \Delta_2\restriction P_2 = (\Delta_1 \cup \Delta_2)\restriction(P_1 \mid P_2)$, since by theorem 3.1.4.i all free names of P_i are in Δ_i.

Case the last rule is WEAK.
Suppose that $\vdash_{()\mu} P \succ \Delta$. The induction hypothesis are that $\mathcal{TA}(P)$ succeeds and that there is a substitution s such that $s\mathcal{TA}(P) = \Delta\restriction P$.
i Directly from the induction hypothesis.
ii Suppose that the application of the last rule yields $\vdash_{()\mu} P \succ \Delta \cdot x:\alpha$, for x not in Δ and α some type scheme. By theorem 3.1.4.i all free names of P are in Δ. Hence x is not free in P and thus $\Gamma\restriction P = (\Gamma \cdot x:\alpha)\restriction P$. □

5 Comparison to Milner's sorting program

Theorem 5.1.2 (Regular Systems)
i Corresponds to the first part of theorem 4.2.1. of [4]. **ii** The proof of theorem 4.2.1. of [4] still holds if u is an element of $\{\mathbf{Subtree}(t) \mid t \in T\}$, T a finite set of trees. □

Theorem 5.1.3 (Sortings and typings)
($\Rightarrow$) Directly from the method of solving the system of equations corresponding to $\mathcal{S}$ and ob (cf. 4.4 of [4].) ($\Leftarrow$) Directly from the method of building the system of equations corresponding to the regular trees in Γ (cf. 4.2.1 of [4].) □

Lemma 5.2.2 (Admissibility of the system FR)
The rule for a located abstraction of the form $a.\lambda x_1 \cdots x_n A$ can be replaced by a deduction of the form,

$$\dfrac{a : s \qquad \dfrac{x_n : s_n \qquad \dfrac{\vdots \qquad \dfrac{x_1 : s_1 \qquad A : ()}{(\lambda x_1)A : (s_1)}}{(\lambda x_1 \cdots x_{n-1})A : (s_1 \cdots s_{n-1})}}{(\lambda x_1 \cdots x_n)A : (s_1 \cdots s_n)}}{a.\lambda x_1 \cdots x_n A : ()} \quad (ob(s) = s_1 \cdots s_n)$$

A similar construction proves the admissibility of the rule for a located concretion. The remaining rules are already in the original system. □

Theorem 5.2.4 (Well-sortedness and well-typedness)
(Outline) **i** From ob and the assignment of names to sorts, build Γ. A deduction of $[\![A]\!]_C$ can be obtained by replacing all occurrences of $\mathbf{0}:()$ in the deduction of $A:()$ by $\vdash_{()\mu} \mathbf{0} \succ \Gamma$ (by using the WEAK rule as many times as needed) and by replacing every occurrence of any other rule in FR by the corresponding one in $\mathrm{TA}_{()\mu}$. The side conditions on compatibility of typings hold from the fact that all typings in the

deduction are subsets of Γ. Conditions $\tilde{x}:\tilde{\rho}$ and $a:(\tilde{\rho})$ in RCPT and EMT rules hold from theorem 5.1.3.

ii We assume bound names under different bindings in P are all distinct and disjoint from the free names. Collect in Γ all type assignments appearing in the deduction of P and, from this typing, build $\mathcal{S}$ and ob. In order to build a deduction of $[\![P]\!]_P$, erase all occurrences of the WEAK and the $(\asymp)$ rules form the deduction of P, and replace occurrences of the other rules by the corresponding rules in FR. Conditions $a:s$ and $\tilde{x}:ob(s)$ in the rules corresponding to RCPT and EMT hold from theorem 5.1.3. $\Box$