Predicative Polymorphism in π-Calculus*

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

Abstract

We present a formulation of the polyadic π-calculus featuring a syntactic category for agents, together with a typing system assigning polymorphic types to agents. The new presentation introduces an operator to express recursion, and an ML-style let-constructor allowing to associate an agent to an agent-variable, and use the latter several times in a program. The essence of the monomorphic type system is the assignment of types to names, and multiple name-type pairs to programs [14]. The polymorphic type system incorporates a form of abstraction over types, and inference rules allowing to introduce and eliminate the abstraction operator. The extended system preserves most of the syntactic properties of the monomorphic system, including subject-reduction and computability of principal typings. We present an algorithm to extract the principal typing of a process, and prove it correct with respect to the typing system. We also study, in the context of π-calculus, some well-known properties of the let-constructor.

1 Introduction

Predicative polymorphism allows to write a piece of code once, and use it in different contexts, with different types, instances of the type assigned to the original code.

The original proposal of the π-calculus [8] uses agent-constants and agent equations to provide for recursion. Agents are processes abstracted on a sequence of names, usually the free names of the process, while equations assign agent-constants to agents. Recursion is obtained by allowing an agent-constant to appear in the right-hand side of its defining equation. A typical program consists of a collection of equations defining agents, and a process instantiating these agents through their associated constants.

This mechanism not only provides for recursion, but also allows a form of polymorphism. In fact, an agent constant can be used to instantiate processes with different *name-structures* for the names in the instantiated process. This name-structure, arising in name-passing calculi by the simple fact that names carry names, is captured by the notion of types for names [2, 7, 11, 13, 14]. Then we are in presence of a case

*To appear in *5th Parallel Architectures and Languages Europe*, LNCS. Springer-Verlag, July 1994.

of predicative polymorphism where names in the instantiated processes have different types, instances of a more general (polymorphic) type for the agent.

The particular formulation of the π-calculus we propose introduces a new syntactic category for agents (cf. [7, 12]), variables over agents (in contrast with constants), and two new constructors for processes. Agents are either processes abstracted on a sequence of names, $(x^n)\,P$, for $x^n = x_1 \cdots x_n$ a sequence of names and P a process, or recursively defined agents, $\mathsf{rec}\,X.A$, binding agent-variable X in agent A. The new process-constructors are agent application Av^n, and an ML-like constructor $\mathsf{let}\,X = A\,\mathsf{in}\,P$, binding agent-variable X in process P.

This new formulation leads to a natural notion of predicative polymorphism. Following Mitchell [9], the word predicative refers to the fact that polymorphism is introduced only after having defined all the base (monomorphic) types. To names we assign types meant to describe the kind of names a name is able to carry: when a name a carries a sequence of names of types $\alpha_1, \ldots, \alpha_n$, we assign type $(\alpha_1 \cdots \alpha_n)_n$ to a. To agents we assign sequences of types: if x^n are the free names in a process P, and α^n their associated types, then to agent $(x^n)\,P$ we assign α^n, and call it a *monomorphic type* for the agent. In this way, we can abstract α^n on some type-variable t to obtain a *polymorphic type* $\forall t.\alpha^n$ for the agent.

We introduce a pair of rules allowing to abstract a given type on a particular type-variable, and to eliminate this from a polymorphic type. The polymorphic system preserves the important properties of the simple system, including the subject-reduction property and the existence and computability of principal typings. An algorithm to extract the principal typing of a process (cf. [1, 5, 16] for λ-calculi, and [14, 15] for name-passing calculi) is presented and proved correct with respect to the typing system. We also discuss some properties of the let-constructor and the relationship between the polymorphic and the monomorphic type systems, results that show remarkable similarities to what happens in ML.

The outline of the paper is as follows. The next section introduces a version of the polyadic π-calculus with variables over agents. Section 3 extends the basic typing assignment system [14] to handle the new constructors, and studies some properties of the system thus obtained. Section 4 introduces the polymorphic typing assignment system, and Section 5 presents an algorithm to extract the principal typing of a process. Section 6 discusses some further issues.

2 Polyadic π-calculus with variables over agents

This section presents a formulation of the polyadic π-calculus, where variables over agents and an explicit recursion operator provide for unbounded computation power. Unlike the original proposal of the calculus [8], we do not rely on auxiliary equations to provide for infinite behaviour, but else bring these equations directly into the syntax of processes, by introducing the notions of agents (cf. [7, 12]) and of variables over agents.

Definition 2.1 (Syntax) Let $\mathbf{N}$ be an infinite set of *names*, and let $a, b, \ldots$ and also $u, v, x, \ldots$ range over $\mathbf{N}$. Sequences of names of the form $x_1 \cdots x_n$, for $n \geq 0$, are abbreviated to x^n. Let $\mathbf{X}$ be an infinite set of *agent-variables*, and let $X, Y, \ldots$ range over $\mathbf{X}$. Let x^n be a sequence of pairwise distinct names. The set $\mathbf{P}$ of *processes* is given by the grammar

$$\begin{array}{rcl} P & ::= & \overline{a}v^n.P \quad | \quad a(x^n).P \quad | \quad P \mid Q \quad | \quad \nu x P \quad | \quad \mathbf{0} \\ & | & Xv^n \quad | \quad Av^n \quad | \quad \mathsf{let}\, X = A \,\mathsf{in}\, P \end{array}$$

where $P, Q, \ldots$ range over $\mathbf{P}$. The set $\mathbf{A}$ of *agents* is given by the grammar

$$A \quad ::= \quad (x^n)\,P \quad | \quad \mathsf{rec}\, X.A$$

where $A, A', \ldots$ range over $\mathbf{A}$. In $(x^n)\,P$, we assume x^n contains all the free names of P.

Processes of the form $\overline{a}v^n.P$ are called *emitters*: they send the sequence of names v^n along name a, and then behave as P. *Receptors* are processes of the form $a(x^n).P$: they receive a sequence of names v^n on name a, and then behave as P with names x^n replaced by v^n. The *concurrent composition* of P and Q, denoted by $P \mid Q$, represents the process composed of P and Q running in parallel. Processes of the form $\nu x P$ restrict the scope of name x to process P. Multiple scope restrictions $\nu x_1 \cdots \nu x_n P$ are abbreviated to $\nu x^n P$. *Inaction*, the process that does nothing, is denoted by $\mathbf{0}$.

Agents can be of two forms: *simply abstracted processes* $(x^n)\,P$ or *recursively abstracted agents* $\mathsf{rec}\, X.A$. As a consequence of the definition, all agents are of the form $\mathsf{rec}\, X_1. \cdots .\mathsf{rec}\, X_k.(x^n)\,P$, for $k \geq 0$; we call n the arity of A, and denote it by $\mathrm{arity}(A)$. A process Av^n applies the sequence of names v^n to agent A, and in particular allows to instantiate a recursive agent, to obtain a recursive process. Similarly, an *applied agent-variable* Xv^n behaves as Av^n, if X is bound to A by a recursive agent or a let declaration. Finally, a process of the form $\mathsf{let}\, X = A \,\mathsf{in}\, P$ allows to associate agent A with agent-variable X, and to use X multiple times in process P.

Receptors, scope restriction and agents are the *name binding operators* of the calculus, binding the free occurrences of x^n (or of x in the case of scope restriction) in the respective bodies P. The *set of free names* in a process P or agent A, denoted by $\mathrm{fn}(P)$ or $\mathrm{fn}(A)$, is defined accordingly. We assume the usual notion of *multiple substitution* of names v^n for the free occurrences of names x^n in a term P, denoted by $P\{v^n/x^n\}$ and defined only if the names in x^n are distinct.

Variables over agents introduce new notions not present in the original π-calculus, namely those of bindings for agent-variables, free agent-variables, and substitution of agents for agent-variables. All these new notions are handled in a quite conventional way. Recursively defined agents and let-declarations constitute the *agent-variable binding operators*: $\mathsf{rec}\, X.A$ binds X in A, and $\mathsf{let}\, X = A \,\mathsf{in}\, P$ binds X in P. The *set of free agent-variables* in a process P or agent A, denoted by $\mathrm{fv}(P)$ or $\mathrm{fv}(A)$, is defined accordingly. Then, $P[X := A]$ denotes the result of substituting A for the free occurrences of X in P.

Structural congruence over processes simplifies the treatment of reduction. Following Milner [6], we define $\equiv$ to be the smallest congruence relation over processes generated by the following rules.

i. $P \equiv Q$ if P is α-convertible to Q

ii. $P \mid Q \equiv Q \mid P, \qquad (P \mid Q) \mid R \equiv P \mid (Q \mid R), \qquad P \mid \mathbf{0} \equiv P$

iii. $\nu x P \mid Q \equiv \nu x(P \mid Q)$ if $x \notin \mathrm{fn}(Q)$

iv. $((x^n)\, P)v^n \equiv P\{v^n/x^n\}$

v. $\mathsf{rec}\, X.A \equiv A[X := \mathsf{rec}\, X.A]$

vi. $\mathsf{let}\, X = A \,\mathsf{in}\, P \equiv P[X := A]$

Notice that rule vi. allows to completely eliminate occurrences of let from processes and agents. The following formulation of reduction relies on the fact that every process in P can be systematically transformed into an equivalent process of the form $\nu u^k \Psi$, where Ψ represents the concurrent composition of (zero or more) emitters, receptors and applied agent-variables.

Definition 2.2 (Reduction relation) *One-step reduction*, denoted by $P \to Q$, is the smallest relation generated by the following rules.

$$\textsc{Struct} \quad \frac{P' \equiv P \quad P \to Q \quad Q \equiv Q'}{P' \to Q'}$$

$$\textsc{Comm} \quad \nu u^k(\overline{a}v^n.P \mid a(x^n).Q \mid \Psi) \to \nu u^k(P \mid Q\{v^n/x^n\} \mid \Psi)$$

Reduction $\twoheadrightarrow$ is the reflexive and transitive closure of one-step reduction.

3 Monomorphic Type Assignment

This section presents the notion of types for names, and a typing system assigning monomorphic types to agents; it concludes with a brief study of the properties of the typing system.

Definition 3.1 (Types for names) Let $\mathbf{V}$ be an infinite set of type-variables, and let $t, t', \ldots$ range over $\mathbf{V}$. The set $\mathbf{T}$ of *types for names* is given by the grammar

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

for $n \geq 0$, where $\alpha, \beta, \ldots$ range over $\mathbf{T}$.

Type assignments to names are expressions $x\colon \alpha$, for any name x and type α. While we assign types to names, to agents we assign sequences of types, and to processes we assign typings. *Typings*, denoted by $\Gamma, \Delta, \ldots$, are sets of type assignments to names where no name appears twice. If $x^n = x_1 \cdots x_n$ is a sequence of distinct names, and $\alpha^n = \alpha_1 \cdots \alpha_n$ is a sequence of types, we write $x^n\colon \alpha^n$ for the typing $\{x_1\colon \alpha_1, \ldots, x_n\colon \alpha_n\}$, and $\Gamma \cdot x^n\colon \alpha^n$ for $\Gamma \cup x^n\colon \alpha^n$, provided names in x^n do not appear in Γ. Compatible typings can be combined by a simple set-union operation.

Definition 3.2 (Typing compatibility) Typings Γ and Δ are *compatible*, denoted by $\Gamma \asymp \Delta$, if $\alpha = \beta$ for all $x\colon \alpha \in \Gamma$ and $x\colon \beta \in \Delta$.

Type assignments to agent-variables are expressions $X\colon \alpha^n$, for any agent-variable X and sequence of types α^n. *Bases*, denoted by $B, B', \ldots$, are sets of type assignments to agent-variables where no agent-variable occurs twice. Similarly to typings we write $B \cdot X\colon \alpha^n$ for $B \cup \{X\colon \alpha^n\}$, provided X does not appear in B.

Typing assignments to processes are formulas $P \triangleright \Gamma$, for any process P and typing Γ, whereas *type assignment to agents* are expressions $A \triangleright \alpha^n$, for any agent A and sequence of types α^n. We write $B \vdash P \triangleright \Gamma$ (respectively $B \vdash A \triangleright \alpha^n$) if the formula $P \triangleright \Gamma$ (respectively $A \triangleright \alpha^n$) is provable from basis B using the axioms and rules of the typing assignment system described below. Whenever $B \vdash P \triangleright \Gamma$ we say P is *typable* and call Γ a *well-typing* for P.

Definition 3.3 (Simple type assignment system TA) TA is defined by the following axioms and rules for processes

$$\textsc{Nil} \quad B \vdash \mathbf{0} \triangleright \emptyset \qquad\qquad \textsc{Rcpt} \quad \frac{B \vdash P \triangleright \Gamma \cdot x^n : \alpha^n}{B \vdash a(x^n).P \triangleright \Gamma \cup \{a : (\alpha^n)_n\}} \quad (\Gamma \asymp a : (\alpha^n)_n)$$

$$\textsc{Emt} \quad \frac{B \vdash P \triangleright \Gamma}{B \vdash \overline{a}v^n.P \triangleright \Gamma \cup \{a : (\alpha^n)_n\}} \quad (v^n : \alpha^n \subseteq \Gamma \text{ and } \Gamma \asymp a : (\alpha^n)_n)$$

$$\textsc{Scop} \quad \frac{B \vdash P \triangleright \Gamma \cdot x : \alpha}{B \vdash \nu x P \triangleright \Gamma} \qquad\qquad \textsc{Comp} \quad \frac{B \vdash P \triangleright \Gamma \qquad B \vdash Q \triangleright \Delta}{B \vdash P \mid Q \triangleright \Gamma \cup \Delta} \quad (\Gamma \asymp \Delta)$$

$$\textsc{Let} \quad \frac{B \vdash A \triangleright \alpha^n \qquad B \cdot X : \alpha^n \vdash P \triangleright \Gamma}{B \vdash \mathsf{let}\, X = A \,\mathsf{in}\, P \triangleright \Gamma} \qquad\qquad \textsc{Abs} \quad \frac{B \vdash P \triangleright x^n : \alpha^n}{B \vdash (x^n)\,P \triangleright \alpha^n}$$

$$\textsc{Weak} \quad \frac{B \vdash P \triangleright \Gamma}{B \vdash P \triangleright \Gamma \cdot x : \alpha}$$

and the following ones for agents

$$\textsc{Var} \quad B \cdot X : \alpha^n \vdash X \triangleright \alpha^n$$

$$\textsc{App} \quad \frac{B \vdash U \triangleright \alpha^n}{B \vdash U v^n \triangleright v^n : \alpha^n} \qquad\qquad \textsc{Rec} \quad \frac{B \cdot X : \alpha^n \vdash A \triangleright \alpha^n}{B \vdash \mathsf{rec}\, X.A \triangleright \alpha^n}$$

where, in the App-rule, U stands for an agent or an agent-variable.

As an example consider a forwarder of a single name, from name a to name b, defined as $\mathsf{rec}\, X.(ab)\,(a(x).\overline{b}x.Xab)$. If t is a type-variable, it is easy to prove that the agent has a type $(t)_1(t)_1$ under the empty basis.

The rest of this section is dedicated to the study of some important properties of the typing system. Most of the properties here presented are simple extensions of those of the typing system for the calculus without agents and agent-variables [14].

The free names in a process constitute the interaction points of the process, and the types of these names describe, in some sense, the interface of the process. Typings for typable processes assign types to all free names in the process and, whenever a process is typable, there is a derivation assigning types to exactly the free names in the process. In particular, this result guarantees that one can always abstract a process on its free-names, via the Abs-rule. Let $\Gamma \upharpoonright P$ denote the restriction of Γ to the free names of P. We call typings of this form *P-typings*.

Lemma 3.4 *If $B \vdash P \triangleright \Gamma$, then the free names of P appear in Γ, and $B \vdash P \triangleright \Gamma \upharpoonright P$.*

Similarly to typings, bases for typable processes contain all the free agent-variables in the process, and there are derivations starting from bases containing exactly the free agent-variables in the process. Let $B \upharpoonright P$ denote the restriction of B to the free agent-variables of P.

Lemma 3.5 *If $B \vdash P \triangleright \Gamma$, then the free agent-variables of P appear in B, and $B \upharpoonright P \vdash P \triangleright \Gamma$.*

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

Lemma 3.6 *Let P and Q be two structural congruent processes not containing* let*. Then, $B \vdash P \triangleright \Gamma$ if and only if $B \vdash Q \triangleright \Gamma$.*

Processes containing let require a more careful treatment. Similarly to what happens in ML, the typability of a let-process implies the typability of its expansion.

Lemma 3.7 *If $B \vdash \mathsf{let}\, X = A \,\mathsf{in}\, P \triangleright \Gamma$, then $B \vdash P[X := A] \triangleright \Gamma$.*

Proof: Suppose $B \vdash \mathsf{let}\, X = A \,\mathsf{in}\, P \triangleright \Gamma$. By the Let-rule we have deductions of $B \vdash A \triangleright \alpha^n$, and of $B \cdot X : \alpha^n \vdash P \triangleright \Gamma$. Replace, in the deduction of $B \cdot X : \alpha^n \vdash P \triangleright \Gamma$, deductions of $B \cdot X : \alpha^n \vdash Xv^n \triangleright v^n : \alpha^n$ by those of $B \vdash Av^n \triangleright v^n : \alpha^n$. □

The converse of the above result does not hold when A is not typable and does not occur in P. We have nevertheless the following results.

Lemma 3.8 *Let* $\mathrm{arity}(A) = n$.

i. If $B \vdash A \triangleright \alpha^n$ and $B \vdash P[X := A] \triangleright \Gamma$, then $B \vdash \mathsf{let}\, X = A \,\mathsf{in}\, P \triangleright \Gamma$.

ii. $B \vdash \mathsf{let}\, X = A \,\mathsf{in}\, P \triangleright \Gamma$ if and only if $B \vdash P[X := A] \mid \nu v^n A v^n \triangleright \Gamma$.[1]

Proof: *i.* Suppose $B \vdash A \triangleright \alpha^n$ and $B \vdash P[X := A] \triangleright \Gamma$. Replace, in the deduction of $B \vdash P[X := A] \triangleright \Gamma$, deductions of $B \vdash Av^n \triangleright v^n : \alpha^n$ by those of $B \cdot X : \alpha^n \vdash Xv^n \triangleright v^n : \alpha^n$ to obtain a deduction of $B \cdot X : \alpha^n \vdash P \triangleright \Gamma$. The result follows by the Let-rule.
ii. ($\Rightarrow$) Suppose $B \vdash \mathsf{let}\, X = A \,\mathsf{in}\, P \triangleright \Gamma$. By Lemma 3.7 we have $B \vdash P[X := A] \triangleright \Gamma$. By the Let-rule there is a sequence of types α^n such that $B \vdash A \triangleright \alpha^n$. The result follows by rules App, Scop and Comp.
($\Leftarrow$) Suppose $B \vdash P[X := A] \mid \nu v^n A v^n \triangleright \Gamma$. Since $\mathrm{fn}(\nu v^n A v^n) = \emptyset$, by the Comp-rule we have that A is typable and $B \vdash P[X := A] \triangleright \Gamma$. The result follows by the first part of this lemma. □

Theorem 3.9 (Subject-reduction) *If $B \vdash P \triangleright \Gamma$ and $P \twoheadrightarrow Q$, then $B \vdash Q \triangleright \Gamma$.*

The converse of subject-reduction does not hold in general [14]. Two kinds of problems may occur during reduction: mismatch on the lengths of the sequences of names transmitted along a name, and mismatch on the length of the sequence of names applied to an abstraction. We say P contains a *possible runtime error*, denoted by $P \in \mathrm{Err}$, if $P \twoheadrightarrow Q$ and *either* $Q \equiv \nu x^k(\overline{a}v^n.Q_1 \mid a(x^m).Q_2 \mid \Psi)$ with $n \neq m$, *or* Q has a subprocess Av^n with $\mathrm{arity}(A) \neq n$.

[1] Cf. the corresponding result in ML [3]: $\mathsf{let}\, x = M \,\mathsf{in}\, N$ has the same typings as $((\lambda y.N)x)[M/x]$.

Corollary 3.10 (Lack of runtime errors) *If P is typable, then $P \notin \text{Err}$.*

We say that Δ is an *instance* of Γ (or that Γ *subsumes* Δ) if there is a substitution s such that $s\Delta \subseteq \Gamma$. In this case we say that Δ is an instance of Γ via substitution s. Instances of well-typings are also well-typings.

Lemma 3.11 *Let Δ be an instance of Γ via s. If $B \vdash P \triangleright \Gamma$, then $sB \vdash P \triangleright \Delta$.*

A *principal typing* for a process P is a well-typing for P that subsumes all other well-typings for P, under a given basis. It is easy to see that the principal typing of a process, when it exists, is unique up to renaming to type-variables. Furthermore, by Lemma 3.4, it contains exactly the free names in the process.

Theorem 3.12 (Principal typing property) *If P is typable under a given basis B, then P has a principal typing under B. Moreover this typing can be effectively computed.*

An algorithm to extract the principal typing of a process in a more general setting, or to detect its non-existence, is presented in Section 5.

4 Polymorphic Type Assignment

This section introduces an extension of the monomorphic type system, by allowing abstraction of a sequence of types on a particular type-variable, and the application of a type to a type-abstraction.

Definition 4.1 (Types for agents) The set of *monomorphic types* is given by

$$\tau \quad ::= \quad \alpha_1 \cdots \alpha_n$$

for $n \geq 0$, and the set of *polymorphic types* is defined by following grammar.

$$\sigma \quad ::= \quad \tau \quad | \quad \forall t.\sigma$$

As a consequence of the definition, universal quantifiers can only occur at the top level of types. A polymorphic type of the form $\forall t_1. \cdots .\forall t_n.\tau$, for $n \geq 0$, is abbreviated to $\forall t^n.\tau$. Type $\forall t^o.\tau$ simply denotes the monomorphic type τ. In the sequel, we assume that the type-variables t^n in a polymorphic type $\forall t^n.\tau$ are pairwise distinct.

We now have a binding for type-variables, namely, in type $\forall t.\sigma$ type-variable t binds the free occurrences of t in σ. We assume the usual notion of substitution of a simple type α for the free occurrences of type-variable t in a polymorphic type σ, denoted by $\sigma[\alpha/t]$. We say a type-variable occurs free in basis B if it occurs free in some type in B.

Let B be a polymorphic basis. We write $B \vdash_\forall P \triangleright \Gamma$ (respectively $B \vdash_\forall A \triangleright \sigma$) is statement $P \triangleright \Gamma$ (respectively $A \triangleright \sigma$) is provable from basis B using the axioms and rules of $\text{TA}_\forall$ below.

Definition 4.2 (Polymorphic type assignment system $\mathbf{TA}_\forall$) $\mathrm{TA}_\forall$ is defined by the rules in TA with the monomorphic type α^n in rules LET and VAR (and only in these) replaced by the polymorphic type σ, by allowing polymorphic bases (but not polymorphic typings), and with the addition of the following rules.

$$\forall\text{-INTRO}\quad \frac{B \vdash A \triangleright \sigma}{B \vdash A \triangleright \forall t.\sigma}\ (t \text{ not free in } B) \qquad \forall\text{-ELIM}\quad \frac{B \vdash U \triangleright \forall t.\sigma}{B \vdash U \triangleright \sigma[\alpha/t]}$$

To continue our example, we can now declare a generic forwarder agent F of type $\forall t.(t)_1(t)_1$, and have two processes, $\mathrm{F}pq$ and $\mathrm{F}uv$, forwarding names of different types, each obtained by eliminating $\forall$ from $\forall t.(t)_1(t)_1$, as in

$$\vdash_\forall \mathsf{let}\ \mathrm{F} = \mathsf{rec}\, X.(ab)\,(a(x)\,.\overline{b}x.Xab)\ \mathsf{in}\ \overline{p}5 \mid \mathrm{F}pq \mid \overline{u}\mathbf{t} \mid \mathrm{F}uv \triangleright$$
$$\{5\colon \mathrm{Int}, p\colon (\mathrm{Int})_1, q\colon (\mathrm{Int})_1, \mathbf{t}\colon \mathrm{Bool}, u\colon (\mathrm{Bool})_1, v\colon (\mathrm{Bool})_1\}$$

where Int and Bool are supposed to be types for (names representing) integer and boolean values.

The REC-rule does not allow recursion variables to be of a polymorphic type. Mycroft proposed an extension of the ML typing system by allowing types for recursion variables to be polymorphic [10], and Kfoury et al. even presented an algorithm to extract the principal type of a term in the extended system [4]. However, the algorithm seems not to be of practical application. We anticipate the same kind of problems raised by Mycroft if we allow recursion variables of a polymorphic type.

All properties of system TA discussed in Section 3 extend to system $\mathrm{TA}_\forall$. We conclude this section with a brief comparison between systems $\mathrm{TA}_\forall$ and TA.

Similarly to ML, what makes $\mathrm{TA}_\forall$ more flexible than TA is the presence of polymorphic agent-variables. The following result express this quite clearly. We say a process P is *closed* if $\mathrm{fv}(P) = \emptyset$. Since let is the only binding for polymorphic agent-variables, closed process without occurrences of let are typable in $\mathrm{TA}_\forall$ if and only if they are typable in TA.

Lemma 4.3

i. Agents have polymorphic types if and only if they have monomorphic types.

ii. Closed processes not containing let *are typable in $\mathrm{TA}_\forall$ if and only if they are typable in TA.*

Proof: *i.* ($\Rightarrow$) Given a deduction ending with a polymorphic type, use the $\forall$-ELIM-rule as may times as needed to obtain a monomorphic type.
($\Leftarrow$) A monomorphic type is also polymorphic.
ii. ($\Rightarrow$) Let P be a process in the conditions of the lemma. Since P does not contain let, the LET-rule cannot be used. Since P is also closed, the VAR-rule can only be used with recursion variables, which are monomorphic. As for the remaining polymorphic rules, it is easy to see that every occurrence of $\forall$-INTRO must be followed by $\forall$-ELIM.
($\Leftarrow$) TA is a subsystem of $\mathrm{TA}_\forall$. □

Nonetheless, there are closed processes typable in $\mathrm{TA}_\forall$ but not in TA. Examples are processes where a let-bound agent-variable X appears twice with different types, generic instances of some polymorphic type assigned to X. Also, there are open terms not containing let that are typable in $\mathrm{TA}_\forall$ but not in TA; just think of the same example with X free instead of let-bound.

5 Typing inference

We present an algorithm to extract the principal typing of a typable process, or to detect that the process is not typable. The algorithm incorporates ideas from that of Vasconcelos and Honda for the simply typed π-calculus [14] (which in turn is inspired in Wand's algorithm for the simply typed λ-calculus [16]), and from that of Damas and Milner for the λ-calculus extended with polymorphic-declarations [1, 5].

The idea is to follow the syntax tree of a given term building a set of equations, and postponing unification as much as possible. The presence of let-subprocesses however forces unification, and only in this case we unify the set of equations.

Given a polymorphic basis B and a process P with all bound names and bound agent-variables renamed to be distinct, the algorithm $\mathcal{V}$ builds a set of equations and a typing. Suppose $\mathcal{V}$ produces a set of equations E and a typing Γ_0, and use a unification algorithm $\mathcal{U}$ on E. If s unifies E, then $s\Gamma_0$ is a well-typing for P under basis sB. Conversely, if P is typable under basis sB, then all its P-typings are of the form $s\Gamma_0 \upharpoonright P$. On the other hand, if either $\mathcal{V}$ or $\mathcal{U}$ fails, then P is not typable.

Similarly, given a renamed agent A and a polymorphic basis B, the algorithm $\mathcal{V}$ builds a set of equations E and a monomorphic type τ. If s unifies E, then $s\tau$ is a type for A under basis sB. Conversely, if A is typable under basis sB, then all its monomorphic types are of the form $rs\tau$, for some substitution r.

Before presenting the algorithm, we need a couple of additional notions [1]. The *closure* of a monomorphic type τ with respect to a basis B, denoted by $\mathrm{clos}_B\,\tau$, binds all type-variables in τ that do not occur free in B. More precisely, if t^n are the type-variables in τ that do not occur in free in B, then $\mathrm{clos}_B\,\tau = \forall t^n.\tau$. An *instance* of a polymorphic type σ of the form $\forall t^n.\tau$, denoted by $\mathrm{inst}\,\sigma$, is a type $\tau[u_1/t_1]\dots[u_n/t_n]$, where $u_1 \dots u_n$ are fresh type-variables.

We write BX for the type associated with agent-variable X in basis B, and Γx^n for the sequence of types associated with names x^n in typing Γ. The algorithm $\mathcal{V}$ first builds a constant typing Γ_0 assigning distinct fresh type-variables to all names in the input process or agent, and then goes on with the code below.

$$
\begin{aligned}
\mathcal{V}(\mathbf{0}, B) &= \emptyset \\
\mathcal{V}(\overline{a}x^n.P, B) = \mathcal{V}(a(x^n).P, B) &= \mathcal{V}(P, B) \cup \{\Gamma_0 a = (\Gamma_0 x^n)_n\} \\
\mathcal{V}(P \mid Q, B) &= \mathcal{V}(P, B) \cup \mathcal{V}(Q, B) \\
\mathcal{V}(\nu x P, B) &= \mathcal{V}(P, B) \\
\mathcal{V}(Xv^n, B) &= \{\Gamma_0 v^n = \mathrm{inst}\, BX\} \\
\mathcal{V}(Av^n, B) &= E \cup \{\Gamma_0 v^n = \tau\} \text{ where } (E, \tau) = \mathcal{V}(A, B) \\
\mathcal{V}(\mathsf{let}\, X = A \,\mathsf{in}\, P, B) &= \mathcal{V}(P, sB \cup \{X\colon \mathrm{clos}_{sB}\, s\tau\}) \text{ where } (E,\tau) = \mathcal{V}(A, B),\, s = \mathcal{U}E \\
\mathcal{V}((x^n)\, P, B) &= (\mathcal{V}(P, B), \Gamma_0 x^n) \\
\mathcal{V}(\mathsf{rec}\, X.(x^n)\, P, B) &= (\mathcal{V}(P, B \cup \{X\colon \Gamma_0 x^n\}), \Gamma_0 x^n)
\end{aligned}
$$

Algorithm $\mathcal{V}$ fails in three cases: in let, when unification fails; in Xv^n, when BX is not defined; in Xv^n (or Av^n), when $\Gamma_0 v^n$ and $\mathrm{inst}\, BX$ (or τ) are sequences of different lengths. It is easy to see that $\mathcal{V}$ always terminates, since every call to $\mathcal{V}$ contains a term strictly smaller than the original.

For terms not containing let, $\mathcal{V}$ is linear on the size of the input [14]. For terms containing let, Kanellakis et al. [3] showed that the Damas-Milner algorithm requires, in the worst case, doubly exponential time to produce its string output, but also remarked that, since ML typing appears to be efficient in practice, the worst case must occur seldom. We conclude the section with a proof of the correctness of $\mathcal{V}$ with respect to $\mathrm{TA}_\forall$.

Theorem 5.1 (Consistency of $\mathcal{V}$)

i. If $\mathcal{UV}(P, B)$ succeeds with s, then $sB \vdash_\forall P \triangleright s\Gamma_0$.

ii. If $\mathcal{V}(A, B)$ succeeds with (E, τ), and $\mathcal{U}E$ succeeds with s, then $sB \vdash_\forall A \triangleright s\tau$.

Proof: By induction on the cases analysed by $\mathcal{V}$, using Lemmas 3.4 and 3.11. □

Theorem 5.2 (Completeness of $\mathcal{V}$)

i. Let $B \vdash_\forall P \triangleright \Gamma$. Then $\mathcal{UV}(P, B)$ succeeds, and Γ is an instance of $s\Gamma_0 \restriction P$ under basis sB, where $s = \mathcal{UV}(P, B)$.

ii. Let $B \vdash_\forall A \triangleright \tau$. Then both $\mathcal{V}(A, B)$ and $\mathcal{U}E$ succeed, and there is a substitution r such that $\tau = rs\tau'$, where $(E, \tau') = \mathcal{V}(A, B)$ and $s = \mathcal{U}E$.

Proof: By induction on the length of deductions. In each case we only prove that $\mathcal{UV}(P, B)$ succeeds; the second part of each item follows from the fact that $\mathcal{U}$ produces a most general unifier. Two cases are detailed below.
When the last rule is EMT, suppose $B \vdash_\forall \overline{a}v^n.P \triangleright \Gamma$. By the EMT-rule (and eventually WEAK), we know that $B \vdash_\forall P \triangleright \Gamma$ and that $v^n\colon \alpha^n \cdot a\colon (\alpha^n)_n \subseteq \Gamma$. By induction, $\mathcal{UV}(P, B) = s$ for some s. Also by induction there is an r such that $rs\Gamma_0 \restriction P \subseteq \Gamma$. Hence $rs\Gamma_0 = rs(\Gamma_0 v^n)_n = (\alpha^n)_n$, and rs unifies $\mathcal{V}(\overline{a}v^n.P, B)$.
When the last rule is LET, suppose $B \vdash_\forall \mathsf{let}\, X = A \,\mathsf{in}\, P \triangleright \Gamma$. By the LET-rule, we have $B \vdash_\forall A \triangleright \mathrm{clos}_B\, \tau$. Use the $\forall$-ELIM-rule as many times as needed to obtain $B \vdash_\forall A \triangleright \tau$. By induction $\mathcal{V}(A, B)$ succeeds. Also by the LET-rule, we have $B \cdot X\colon\ \mathrm{clos}_B\, \tau \vdash_\forall P \triangleright \Gamma$. By Lemma 3.11, we have $sB \cdot X\colon\ \mathrm{clos}_{sB}\, s\tau \vdash_\forall P \triangleright \Gamma$. The result follows by induction.
□

6 Further Issues

It is well known that recursion can be replaced by replication. In fact, Milner proposed a formulation of the π-calculus that dispenses agent-constants and agent defining equations in favour of replication [6]. A replicated process of the form $!P$ is meant to denote an unbounded number of copies of P running in parallel, captured by the structural congruence rule $!P \equiv P \mid !P$. The "obvious" rule for $!P$ is the following [14].

$$\textsc{Repl} \quad \frac{B \vdash P \triangleright \Gamma}{B \vdash\, !P \triangleright \Gamma}$$

A recursive process $(\mathsf{rec}\, X.(x^n)\, P)v^n$ can be transformed into a replicated process

$$\nu c(\overline{c}v^n \mid !c(x^n)\,.P') \qquad\qquad (c \text{ fresh}),$$

where P' is obtained from P by replacing occurrences of Xu^n by $\overline{c}u^n$ (an abbreviation for $\overline{c}u^n.\mathbf{0}$). On the other hand, replication can be eliminated in favour of recursion. A replicated process $!P$ can be transformed into

$$(\mathsf{rec}\, X.(\varepsilon)\,(P \mid X))\varepsilon \qquad\qquad (X \text{ fresh}),$$

where ε represents the empty sequence of names. Recursive processes and their replicated counterparts have exactly the same typings.

Lemma 6.1 *Let P be a recursive process and P' its replicated counterpart. Then, $B \vdash_\forall P \rhd \Gamma$ if and only if $B \vdash_\forall P' \rhd \Gamma$.*

Proof: ($\Rightarrow$) Let $\nu c(\overline{c}v^n \mid !c(x^n).P')$ be the replicated form of $(\mathsf{rec}\, X.(x^n)\, P)v^n$, and suppose $B \vdash_\forall (\mathsf{rec}\, X.(x^n)\, P)v^n \rhd \Gamma$. Since agent-variables bound by rec are monomorphic, replace in the deduction of $B \vdash_\forall P \rhd \Delta$ occurrences of deductions of $B \cdot X : \tau \vdash_\forall Xu^n \rhd u^n : \tau$ by the deduction of $B \vdash_\forall \overline{c}u^n \rhd u^n : \tau \cdot c : (\tau)_n$. Since c is not free in P we have a deduction $B \vdash_\forall P' \rhd \Delta \cdot c : (\tau)_n$. The result follows by rules Rcpt, Repl, Comp and Scop, in this order.
($\Leftarrow$) Let $(\mathsf{rec}\, X.(\varepsilon)\,(P \mid X))\varepsilon$ be the replicated form of $!P$, and suppose $B \vdash_\forall !P \rhd \Gamma$. By the Repl-rule we have a deduction of $B \vdash_\forall P \rhd \Gamma$. We also have a deduction of $B \cdot X : \varepsilon \vdash_\forall X\varepsilon \rhd \emptyset$, and since X is not free in P, we have a deduction of $B \cdot X : \varepsilon \vdash_\forall P \rhd \Gamma$. The result follows by rules Comp, Abs and Rec, in this order. □

Notice that the above proof relies on the fact that types associated with recursion variables are monomorphic. We conclude the section by showing that $\mathrm{TA}_\forall$ indeed increases the flexibility of typing, in the sense there are processes typable in $\mathrm{TA}_\forall$ whose "corresponding" processes in the basic system without agent-variables and let-declarations [14] are not typable.

There are ML terms of the form $\mathsf{let}\ x = M\ \mathsf{in}\ N$ that can be typed, whereas the corresponding function application $(\lambda x.N.)M$ cannot. In the (first-order) π-calculus we have no form of application of processes to agents, but we have a device by which we can simulate the sharing of a process (cf. [6, 12]). Using the replication operator introduced above, the idea of sharing an abstraction $(x^n)\,P$ in a process Q via an agent-variable X can be materialized into the process

$$\nu c(!c(x^n).P \mid Q') \qquad\qquad (c \text{ fresh}),$$

where Q' is obtained from Q by replacing occurrences of Xu^n by $\overline{c}u^n$. Notice the usage of replication for an effective sharing of process P when X occurs more than once in Q.

Similarly to ML, the process $\mathsf{let}\ X = (x^n)\,P\ \mathsf{in}\ Q$ may be typable, whereas process $\nu c(!c(x^n).P \mid Q')$ may not be so, for X may occur in Q with different types (instances of a polymorphic type for $(x^n)\,P$), whereas occurrences of c in Q' must all have the same type. This is a direct consequence of the decision of not having polymorphic types for names. Concentrating on name polymorphism instead of agent polymorphism, Turner presented an impredicative polymorphic system for the π-calculus [13] that should be able to type the above process. Nonetheless, typing inference in impredicative systems is still an open problem.

Acknowledgements. I would like to thank Professor Tokoro for support and encouragement, and Kohei Honda for discussions on the enhanced flexibility of the polymorphic system.

References

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

[2] Kohei Honda. Types for dyadic interaction. In *4th International Conference on Concurrency Theory*, volume 715 of *LNCS*, pages 509–523. Springer-Verlag, August 1993.

[3] Paris C. Kanellakis, Harry G. Mairson, and John C. Mitchell. Unification and ML type reconstruction. In *Computational Logic, Essays in Honor of Alan Robinson*, pages 444–478. MIT Press, 1991.

[4] A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. A proper extension of ML with an effective type-assignment. In *15th ACM Symposium on Principles of Programming Languages*, pages 58–69, 1988.

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

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

[7] Robin Milner. The polyadic π-calculus: a tutorial. ECS-LFCS 91–180, University of Edinburgh, October 1991.

[8] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I and II. *Information and Computation*, 100:1–77, 1992. Also as Tech. Rep. ECS-LFCS 89-85/86, University of Edinburgh.

[9] John C. Mitchell. Type systems for programming languages. In *Handbook of Theoretical Computer Science*, pages 366–358. Elsevier Science Publishers B.V., 1990.

[10] A. Mycroft. Polymorphic type schemes for functional programming. In *9th ACM Symposium on Principles of Programming Languages*, 1984.

[11] Benjamin Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. In *8th IEEE Symposium on Logic in Computer Science*, June 1993.

[12] Davide Sangiorgi. *Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms*. PhD thesis, University of Edinburgh, 1992.

[13] David N. Turner. A π-calculus sorting discipline. University of Edinburgh, December 1992.

[14] Vasco T. Vasconcelos and Kohei Honda. Principal typing-schemes in a polyadic π-calculus. In *4th International Conference on Concurrency Theory*, volume 715 of *LNCS*, pages 524–538. Springer-Verlag, August 1993. Also as Keio University Report CS 92-002.

[15] Vasco T. Vasconcelos and Mario Tokoro. A typing system for a calculus of objects. In *1st International Symposium on Object Technologies for Advanced Software*, volume 742 of *LNCS*, pages 460–474. Springer-Verlag, November 1993.

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