数理解析研究所講究録
第 851 巻 1993 年 78-87

A Predicative Polymorphic Type System for a Calculus of Objects

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

July 20, 1993

Abstract

The present paper introduces an untyped calculus of intended objects to capture intrinsic aspects of concurrent objects communicating via asynchronous message passing, together with a typing system assigning typings to terms in the calculus. Types meant to describe the kind of messages an object may receive are assigned to the free names in a program, resulting in a scenario where a program is assigned multiple name-type pairs. Then, a ML-like let declaration is introduced together with an extension of the monomorphic typing assignment system, which allows to declare a term of a polymorphic type and use it multiple times with different types, instances of the type of the declared term. The system enjoys desirable properties, such as subject-reduction which implies that programs that comply to the typing discipline do not suffer from runtime errors, as well as the existence and computability of principal types. Furthermore, we present an efficient algorithm to extract the principal typing of a term.

Introduction

Most of the attempts to introduce some type discipline into object-oriented languages start from lambda-calculus, by extending this with some kind of records. There are several limitations to this approach, mainly deriving from the fact that objects are not extensions of functions. In particular, objects do not necessarily present an input-output behavior; objects usually communicate by asynchronous message passing (instead of function application); objects do maintain a state (in contrast with the stateless nature of functions), and objects may run concurrently.

Inspired by Milner's polyadic π-calculus [6], Honda's ν-calculus [4] and Hewitt's actor model [2], we presented in [11] a basic object-calculus where the notions of objects, asynchronous messages and concurrency are primitive, and introduced a type discipline along the lines of Honda [3] and Vasconcelos and Honda [10] for the (untyped) calculus, enjoying the properties that programs that verify the discipline will never run into errors of the kind "message not understood", and that there is a computable notion of principal typings from which all typings that make a term well-typed can be derived.

The present work extends the basic system by introducing in the calculus variables over terms and a ML-like let declaration, together with a notion of ***predicative polymorphism*** in the type system. Following Mitchell [7], the word predicative refers to the fact that polymorphism is introduced only after having defined all the base (monomorphic) types. The result is an extended calculus of object where we can have polymorphic term declarations, in the sense that a term can be declared to have a polymorphic type, and used in a program multiple times with different types, instances of the type of the declared term.

All the basic syntactic properties of the monomorphic system extend to the polymorphic setting, in particular the subject reduction property and the existence of an algorithm to extract the principal typing of a term. Like in ML, the polymorphic system is strictly more general than its monomorphic counterpart, in the sense that there are let terms of the form **let** $X(\tilde{x}) = P$ **in** Q that can be typed whereas the corresponding term where P is shared in Q via variable X is not typable.

Following [3, 10], types are assigned to names, and not to processes, the latter being assigned multiple name-type pairs, constituting a typing for the process. Types are built from variables by means of a single constructor $[l_1 : \tilde{\alpha}_1, \ldots l_n : \tilde{\alpha}_n]$, representing a name associated with an object capable of receiving messages labeled with l_i carrying sequences of names of types $\tilde{\alpha}_i$, for $1 \leq i \leq n$. A typing assignment system assigns a type to each free name in a term, thus specifying in some sense the interface of the process. To describe objects about which we do not have complete information (e.g. when we only know part of their methods) we use constraints on the types type variables may be substituted for, in the form of Ohori's kinds [8].

The outline of the paper is as follows. The next section introduces the basic calculus, and section 2 the corresponding monomorphic type assignment system. Then, section 3 introduces a form of let declaration in the calculus, and section 4 the polymorphic type assignment system. Section 5 studies some of the properties of the typing systems, and based on the results proposes a simplified system, which will be the basis of the typing inference algorithm to be described in Section 6. Section 7 discusses and compares the present calculus and its typing system to related systems. Finally, the last section concludes the paper.

1 The Basic Calculus

This section introduces the basic calculus to the extent needed for typing considerations. Since we are going to need variables over terms any way, we try a variant of [11] where explicit recursion is built from variables over terms. Structural congruence on terms caters for equivalence over concrete syntax and, together with normal forms and message application, make the formulation of the transition relation quite concise.

Syntax. Simple terms **P** are built from an infinite set of names **N**, an infinite set of term-variables **X** and a set of labels **L**, by means of six constructors,

$$\begin{array}{l} a \lhd l(\tilde{v}) \\ a \rhd [l_1(\tilde{x}_1).P_1 \;\&\cdots\&\; l_n(\tilde{x}_n).P_n] \\ (\nu x)P \\ P, Q \\ X(\tilde{v}) \\ \mathbf{rec}\, X(\tilde{v}).a \rhd [l_1(\tilde{x}_1).P_1 \;\&\cdots\&\; l_n(\tilde{x}_n).P_n] \end{array}$$

where $a, b, \ldots$ and $v, x, \ldots$ are names in **N**; $\tilde{v}, \tilde{x}, \ldots$ are sequences of names in $\mathbf{N}^*$; $X, Y, \ldots$ are term-variables in **X**; $l, m, \ldots$ are labels in **L** and $P, Q, \ldots$ are arbitrary terms in **P**. In an object of the from $a \rhd [l_1(\tilde{x}_1).P_1 \;\&\cdots\&\; l_n(\tilde{x}_n).P_n]$, we assume the names in $\tilde{x}_i$ and the labels l_i pairwise distinct, for $n \geq 0$ and $1 \leq i \leq n$.

Intuitively, a term of the form $a \lhd l(v_1 \cdots v_n)$ denotes an asynchronous message directed to an object located at name a, selecting a method labeled with l, and carrying a sequence of names $v_1 \cdots v_n$ as actual parameters. Objects are terms of the form $a \rhd [l_1(\tilde{x}_1).P_1 \;\&\cdots\&\; l_n(\tilde{x}_n).P_n]$ comprising an object location or identifier a and an unordered collection of methods $l_i(\tilde{x}_i).P_i$. Object methods are identified by a label l and parameterized by a sequence of names $\tilde{x}$. Intuitively, a method of the form $l(\tilde{x}).P$ matches a communication $l(\tilde{v})$ and behaves as P with occurrences of names in $\tilde{x}$ replaced by those in $\tilde{v}$.

Scope restriction allows for local object creation avoiding unwanted communications with the exterior. If x is a name and P is a term, then $(\nu x)P$ denotes the restriction of x to the scope defined by P. Multiple name restrictions on a term $(\nu x_1)\cdots(\nu x_n)P$ will be written $(\nu\tilde{x})P$. A term of the form P, Q denotes the term composed of P and Q running concurrently.

Intuitively, term-variables stand for terms. If X is a term-variable representing a term P with free names $x_1, \ldots x_n$, then $X(v_1 \cdots v_n)$ denotes the term $P\{v_1 \cdots v_n / x_1 \cdots x_n\}$, that is, the term P with names $x_1, \ldots x_n$ replaced by names $v_1, \ldots v_n$. By allowing X to occur free in $P_1, \ldots P_n$, the constructor $\mathbf{rec}\, X(\tilde{v}).a \rhd [l_1(\tilde{x}_1).P_1 \;\&\cdots\&\; l_n(\tilde{x}_n).P_n]$ allows for recursive object definition. For succinctness we will often write $\mathbf{rec}\, X(\tilde{v}).P$ instead of the more verbose form $\mathbf{rec}\, X(\tilde{v}).a \rhd [l_1(\tilde{x}_1).P_1 \;\&\cdots\&\; l_n(\tilde{x}_n).P_n]$, but one should keep in mind that we only allow recursion over objects. Inaction is a convenient derived constructor. Denoted by **0**, it represents the process which does nothing, and is defined as $(\nu x)\; x \rhd []$. The length of the sequence of names $\tilde{x}$ is denoted by $len(\tilde{x})$, and the set of names occurring in $\tilde{x}$ by $\{\tilde{x}\}$.

Semantics. Methods in objects and scope restriction are the binding operators in the calculus. The set of *free names* in term P, denoted $\mathcal{FN}(P)$, is inductively defined by the following rules.[1]

$$\begin{array}{rcl} \mathcal{FN}(a \lhd l(\tilde{v})) & = & \{a\} \cup \{\tilde{v}\} \\ \mathcal{FN}(a \rhd [l_1(\tilde{x}_1).P_1 \;\&\cdots\&\; l_n(\tilde{x}_n).P_n]) & = & \\ \multicolumn{3}{c}{\{a\} \cup \mathcal{FN}(P_1) \setminus \{\tilde{x}_1\} \cup \cdots \cup \mathcal{FN}(P_n) \setminus \{\tilde{x}_n\}} \\ \mathcal{FN}((\nu x)P) & = & \mathcal{FN}(P) \setminus \{x\} \\ \mathcal{FN}(P, Q) & = & \mathcal{FN}(P) \cup \mathcal{FN}(Q) \\ \mathcal{FN}(X(\tilde{v})) & = & \{\tilde{v}\} \\ \mathcal{FN}(\mathbf{rec}\, X(\tilde{v}).P) & = & \{\tilde{v}\} \cup \mathcal{FN}(P) \end{array}$$

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

There is also a *binding on term-variables*, namely, term-variable X occurs bound in $\mathbf{rec}\, X(\tilde{v}).P$. The set of *free variables* in term P, denoted $\mathcal{FV}(P)$, is in-

[1] '$\setminus$' binds tighter than '$\cup$'.

ductively defined by the following rules.

$$\begin{aligned}
\mathcal{FV}(a \lhd l(\tilde{v})) &= \emptyset \\
\mathcal{FV}(a \rhd [l_1(\tilde{x}_1).P_1 \ \& \cdots \& \ l_n(\tilde{x}_n).P_n]) &= \\
&\quad \mathcal{FV}(P_1) \cup \cdots \cup \mathcal{FV}(P_n) \\
\mathcal{FV}((\nu x)P) &= \mathcal{FV}(P) \\
\mathcal{FV}(P, Q) &= \mathcal{FV}(P) \cup \mathcal{FV}(Q) \\
\mathcal{FV}(X(\tilde{v})) &= \{X\} \\
\mathcal{FV}(\mathbf{rec}\, X(\tilde{v}).P) &= \mathcal{FV}(P) \setminus \{X\}
\end{aligned}$$

A term is said to be *closed* if it contains no free variables. *Substitution of a term-variable* X by a term Q with free names $\tilde{x}$, in a term P, denoted by $P[Q/X]_{\tilde{x}}$ is the result of substituting $Q\{\tilde{v}/\tilde{x}\}$ for every free occurrence of the form $X(\tilde{v})$ in P, and changing bound variables to avoid capture of free variables. The precise definition is by induction on P and holds only when $len(\tilde{x}) = len(\tilde{v})$, for all sub-terms of the form $X(\tilde{v})$ in P.

Structural congruence provides for syntactic equivalence over terms, simplifying the treatment of reduction. It is the smallest congruence relation defined by the following rules.

$$\begin{array}{l}
P \equiv Q \text{ whenever } P \ \alpha\text{-convertible to } Q \\
P, Q \equiv Q, P \text{ and } P, (Q, R) \equiv (P, Q), R \\
a \rhd [l(\tilde{x}).P \ \& \ m(\tilde{y}).Q] \equiv a \rhd [m(\tilde{y}).Q \ \& \ l(\tilde{x}).P] \\
(\nu x)P, Q \equiv (\nu x)(P, Q) \text{ whenever } x \notin \mathcal{FN}(Q) \\
\mathbf{rec}\, X(\tilde{v}).P \equiv P[\mathbf{rec}\, X(\tilde{v}).P/X]_{\tilde{v}}
\end{array}$$

Normal forms further simplify the treatment of reduction. Every closed term in $\mathbf{P}$ can be transformed into an equivalent term of the form,

$$(\nu \tilde{u})(P_1, \ldots, P_m)$$

for some $m \geq 0$, where $P_1, \ldots P_m$ denote messages or objects.

Message application constitutes the basic mechanism of the calculus, and represents the reception of a message by an object, followed by the selection of the appropriate method, the substitution of the message contents by the method's formal parameters, and the execution of the method body. The application of the communication $l(\tilde{v})$ of some message to a collection of methods $[l_1(\tilde{x}_1).P_1 \ \& \cdots \& \ l_n(\tilde{x}_n).P_n]$ is defined by,

$$[l_1(\tilde{x}_1).P_1 \ \& \cdots \& \ l_n(\tilde{x}_n).P_n] \bullet l(\tilde{v}) \to P_k\{\tilde{v}/\tilde{x}_k\}$$

whenever $l = l_k \in \{l_1, \ldots l_n\}$ and the lengths of $\tilde{v}$ and $\tilde{x}_k$ match.

Reduction models the computing mechanism of the calculus. By using structural congruence, normal forms and message application, it can be concisely defined. *One-step reduction*, denoted by $\to$, is the smallest relation generated by the following rules.

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

$$\frac{M \bullet C \to P}{(\nu \tilde{x})(\partial, a \lhd C, a \rhd M, \partial') \to (\nu \tilde{x})(\partial, P, \partial')}$$

where ∂ and ∂' represent concurrent composition of messages and objects, C is a communication of the form $l(\tilde{v})$ and M is a collection of methods of the form $[l_1(\tilde{x}_1).P_1 \ \& \cdots \& \ l_n(\tilde{x}_n).P_n]$. The *reduction* relation $\twoheadrightarrow$ is the reflexive and transitive closure of one-step reduction.

One of the simplest stateful objects is a buffer cell. Such an object has two methods, read and write, intended to read and write a value in the cell. Together with a read request comes a name intended to receive the value the cell is holding. Here is a possible definition.

$$\begin{array}{c}\mathbf{rec}\ Cell(sv).\ s \rhd [\mathsf{read}(r).\ r \lhd \mathsf{value}(v),\ Cell(sv)\ \& \\ \mathsf{write}(u).\ Cell(su)]\end{array}$$

2 Monomorphic Typing Assignment

This section introduces the notion of types for names, a system to assign typings (i.e. sets of name-type pairs) to terms and studies some of the properties of the typing system.

Types and Type Assignment.

The set of types $\mathbf{T}$ is built from an infinite set of type-variables $\mathbf{V}$, by means of a single constructor,

$$[l_1 : \tilde{\alpha}_1, \ldots l_n : \tilde{\alpha}_n]$$

where $\tilde{\alpha}_1, \ldots \tilde{\alpha}_n$ are sequences of types in $\mathbf{T}^*$, and $l_1, \ldots l_n$ are pairwise distinct labels in $\mathbf{L}$. Informally, an expression of the form $[l_1 : \tilde{\alpha}_1, \ldots l_n : \tilde{\alpha}_n]$ is intended to denote some collection of names identifying objects containing n methods labeled with $l_1, \ldots l_n$ and whose arguments of method l_i belong to types $\tilde{\alpha}_i$.

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

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

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

Type assignment to term-variables are formulas $X : \tilde{\alpha}$, for X a term-variable and $\tilde{\alpha}$ a sequence of types in $\mathbf{T}^*$. *Bases* are sets of formulas of the form $\{X_1 : \tilde{\alpha}_1, \dots X_n : \tilde{\alpha}_n\}$ where no two formulas have the same term-variable as subject. $B, B', \dots$ will range over bases.

Kinds and Kind Assignment.

Kinds describe constraints on the substitution of type variables. The set of kinds $\mathbf{K}$ is given by all expressions of the form,

$$\langle l_1 : \tilde{\alpha}_1, \dots l_n : \tilde{\alpha}_n \rangle$$

where $l_1, \dots l_n$ are pairwise distinct labels in $\mathbf{L}$ and $\tilde{\alpha}_1, \dots \tilde{\alpha}_n$ are sequences of types in $\mathbf{T}^*$, for $n \geq 0$. $k, k', \dots$ will range over $\mathbf{K}$.

Intuitively, a kind of the form $\langle l_1 : \tilde{\alpha}_1, \dots l_n : \tilde{\alpha}_n \rangle$ denotes the subset of types containing (at least) the components $l_1 : \tilde{\alpha}_1, \dots l_n : \tilde{\alpha}_n$.

Every type variable must be assigned a kind. *Kind assignments* are expressions $t : k$, for t a type variable and k a kind. *Kindings* are acyclic sets of kind assignments[2] where no two assignments have the same type variable as subject. $K, K', \dots$ will range over kindings. We say a type α *has a kind k under a kinding K*, denoted by $K \vdash \alpha : k$, if and only if,

$$K \vdash [l_1 : \tilde{\alpha}_1, \dots l_n : \tilde{\alpha}_n, \dots] : \langle l_1 : \tilde{\alpha}_1, \dots l_n : \tilde{\alpha}_n \rangle$$

$$K \cdot t : \langle l_1 : \tilde{\alpha}_1, \dots l_n : \tilde{\alpha}_n, \dots \rangle \vdash t : \langle l_1 : \tilde{\alpha}_1, \dots l_n : \tilde{\alpha}_n \rangle$$

When $\tilde{\alpha}$ is the sequence of types $\alpha_1 \cdots \alpha_n$ and $\tilde{k}$ is the sequence of kinds $k_1 \cdots k_n$, we write $K \vdash \tilde{\alpha} : \tilde{k}$ to mean $K \vdash \alpha_1 : k_1, \dots K \vdash \alpha_n : k_n$.

Typing Assignment to Terms.

The following notation simplifies the treatment of the typing assignment system. Let $\tilde{x} = x_1 \cdots x_n$ be a sequence of names, $\tilde{\alpha} = \alpha_1 \cdots \alpha_n$ a sequence of types and Γ a typing. Then, $\{\tilde{x} : \tilde{\alpha}\}$ denotes the typing $\{x_1 : \alpha_1, \dots x_n : \alpha_n\}$ and $\Gamma \cdot \tilde{x} : \tilde{\alpha}$ denotes the typing $\Gamma \cup \{\tilde{x} : \tilde{\alpha}\}$, provided names in $\tilde{x}$ do not occur in Γ. Similarly, $B \cdot X : \tilde{\alpha}$ denotes the basis $B \cup \{X : \tilde{\alpha}\}$, provided the term-variable X does not occur in basis B. Finally, $\Gamma / \tilde{x}$ denotes the typing Γ with formulas with subjects in $\tilde{x}$ removed.

We will write $K, B \vdash P \Rightarrow \Gamma$ if the statement $P \Rightarrow \Gamma$ is provable from kinding K and basis B, using the rules and axioms of TA in figure 1. Whenever $K, B \vdash P \Rightarrow \Gamma$ for some kinding K and basis B and typing Γ, we say P is *typable* in TA, and call the pair (K, Γ) a *well kinded-typing* for P (under basis B).

Whenever a term P is typable, there exists a TA derivation starting form a basis containing only the free variables of P and producing a typing containing only assignments on the free names of P. If B is a basis, let $B \lceil P$ be the restriction of B to the free variables of P. We shall call bases of this form *P-bases*. Similarly, if Γ is a typing, let $\Gamma \lceil P$ be the restriction of Γ to the free names of P. Typings of this form shall be called *P-typings*.

Lemma 2.1 *If $K, B \vdash P \Rightarrow \Gamma$, then every free variable of P appears in B, every free name of P appears in Γ, and there is a derivation of $K, B \lceil P \vdash P \Rightarrow \Gamma \lceil P$.*

The following Lemma ensures that structural congruent terms have the same typings.

Lemma 2.2 *If $K, B \vdash P \Rightarrow \Gamma$ and $P \equiv Q$, then $K, B \vdash Q \Rightarrow \Gamma$.*

The following fundamental property of the type assignment system TA ensures that the typing of a term does not change as it is reduced and is closely related with the lack of runtime errors.

Theorem 2.3 (Subject Reduction)
If $K, B \vdash P \Rightarrow \Gamma$ and $P \twoheadrightarrow Q$, then $K, B \vdash Q \Rightarrow \Gamma$.

Notice that the converse of subject-reduction does not hold, since non typable terms can be reduced to typable ones (e.g. $a \triangleleft l(a), a \triangleright [l(x).0] \twoheadrightarrow 0$), and also because free-names may be lost in the course of reduction (e.g. $\vdash 0 \Rightarrow \emptyset$ and $a \triangleleft l(v), a \triangleright [l(x).0] \twoheadrightarrow 0$ but $\not\vdash a \triangleleft l(v), a \triangleright [l(x).0] \Rightarrow \emptyset$). Also due to the loss of free names during reduction, if $K, B \vdash P \Rightarrow \Gamma$, $P \twoheadrightarrow Q$, and $K, B \vdash Q \Rightarrow \Delta$, then $\Delta \lceil Q \subseteq \Gamma \lceil P$.

A consequence of the subject-reduction property is that typable programs will not run into type errors during execution. We say P contains a possible *runtime error*, and write $P \in \textsc{Err}$, if there exists a term Q such that $P \twoheadrightarrow Q \equiv (\nu \tilde{u})(\partial, a \triangleleft l(\tilde{v}), a \triangleright [l_1(\tilde{x}_1).P_1 \mathbin{\&} \cdots \mathbin{\&} l_n(\tilde{x}_n).P_n], \partial')$ and $[l_1(\tilde{x}_1).P_1 \mathbin{\&} \cdots \mathbin{\&} l_n(\tilde{x}_n).P_n] \bullet l(\tilde{v})$ is not defined; that is, either $l \notin \{l_1, \dots l_n\}$ or else $l = l_k \in \{l_1, \dots l_n\}$ but $\mathit{len}(\tilde{v}) \neq \mathit{len}(\tilde{x}_k)$.

Corollary 2.4 (Lack of runtime errors)
If P is typable, then $P \notin \textsc{Err}$.

Consider the buffer cell at the end of the previous section. Since method write expects a name of any type t (the type of the value the cell holds), and method read expects a name capable of receiving *at least* a message of type value : t, a well kinded-typing for the cell is given by,

$$(\{t : \langle\rangle, u : \langle \mathsf{value} : t \rangle\}, \{s : [\mathsf{read} : u, \mathsf{write} : t],\ v : t\})$$

Then, substituting u for the type $[\mathsf{value} : t, \mathsf{print} : \alpha]$ we have that,

$$\{s : [\mathsf{read} : [\mathsf{value} : t, \mathsf{print} : \alpha], \mathsf{write} : t],\ v : t\}$$

[2] A cycle in a set of kind assignments is a sequence of elements $t_1 : k_1, \dots t_n : k_n$ such that t_{i+1} occurs in k_i and t_1 occurs in k_n, for $n \geq 1$.

$$\textsc{Msg}\quad \frac{K,B \vdash \beta : \langle l : \tilde{\alpha}\rangle}{K,B \vdash a \triangleleft l(\tilde{v}) \Rightarrow \{\tilde{v}:\tilde{\alpha}, a:\beta\}} \quad (\{a:\beta\} \asymp \{\tilde{v}:\tilde{\alpha}\})$$

$$\textsc{Obj}\quad \frac{K,B \vdash P_i \Rightarrow \Gamma_i \cdot \tilde{x}_i : \tilde{\alpha}_i}{K,B \vdash a \triangleright [l_1(\tilde{x}_1).P_1 \;\&\cdots\&\; l_n(\tilde{x}_n).P_n] \Rightarrow \{a:[l_1:\tilde{\alpha}_1, \ldots l_n:\tilde{\alpha}_n]\} \bigcup_i \Gamma_i} \quad \begin{array}{r}(\{a:[l_1:\tilde{\alpha}_1,\ldots l_n:\tilde{\alpha}_n]\} \asymp \Gamma_i,\\ \Gamma_i \asymp \Gamma_j, 1 \le i,j \le n)\end{array}$$

$$\textsc{Scop}\quad \frac{K,B \vdash P \Rightarrow \Gamma}{K,B \vdash (\nu x)P \Rightarrow \Gamma/x} \qquad \textsc{Conc}\quad \frac{K,B \vdash P \Rightarrow \Gamma \quad K,B \vdash Q \Rightarrow \Delta}{K,B \vdash P,Q \Rightarrow \Gamma \cup \Delta} \quad (\Gamma \asymp \Delta)$$

$$\textsc{Var}\quad K,B \cdot X:\tilde{\alpha} \vdash X(\tilde{v}) \Rightarrow \{\tilde{v}:\tilde{\alpha}\} \qquad \textsc{Rec}\quad \frac{K,B \cdot X:\tilde{\alpha} \vdash P \Rightarrow \Gamma \cdot \tilde{v}:\tilde{\alpha}}{K,B \vdash \mathbf{rec}\, X(\tilde{v}).P \Rightarrow \Gamma \cdot \tilde{v}:\tilde{\alpha}}$$

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

Figure 1: Monomorphic typing assignment system

is also a well-typing for the cell. However, the typing,

$$\{s:[\mathsf{read}:[\mathsf{value}:t], \mathsf{write}:t, \mathsf{think}:u],\ v:t\}$$

is not acceptable since it would allow us to compose *Cell*(sv) with $s\triangleleft\mathsf{think}(x)$, which would surely run into a type error.

3 Term Declaration

This section introduces a form of term declaration in the basic calculus. In particular, we obtain a form of *object class* declaration from which particular objects can be instantiated.

Syntax. The set $\mathbf{P}_{\mathrm{let}}$ is built by adding to the syntax of $\mathbf{P}$ in section 1 the following constructor.

$$\mathbf{let}\ X(\tilde{x}) = P\ \mathbf{in}\ Q$$

where X does not occur free in P and the names in $\tilde{x}$ are distinct and constitute exactly the free names of P. A declaration of this form allows to define a term P once and use it several times in Q, each one with different instance names.

Semantics. We now have one more binding on names, namely, $\mathbf{let}\ X(\tilde{x}) = P\ \mathbf{in}\ Q$ binds names $\{\tilde{x}\}$ in P, which justifies extending the definition of free names with the following rule.

$$\mathcal{FN}(\,\mathbf{let}\ X(\tilde{x}) = P\ \mathbf{in}\ Q) \;=\; \mathcal{FN}(Q)$$

Notions of free and bound names, substitution and simultaneous substitution, as well as α-conversion extend easily. We also have one more binding on term-variables, namely, the term-variable X occurs bound in $\mathbf{let}\ X(\tilde{x}) = P\ \mathbf{in}\ Q$, from which we extend the definition of free variables in a term with the following rule.

$$\mathcal{FV}(\,\mathbf{let}\ X(\tilde{x}) = P\ \mathbf{in}\ Q) = \mathcal{FV}(P) \cup \mathcal{FV}(Q) \setminus \{X\}$$

The semantics of the new constructor is accounted for by an additional rule in the definition of structural congruence.

$$\mathbf{let}\ X(\tilde{x}) = P\ \mathbf{in}\ Q \;\;\equiv\;\; Q[P/X]_{\tilde{x}}$$

Normal forms can be proved to exist for closed terms in $\mathbf{P}_{\mathrm{let}}$, and so the reduction relation defined in section 1 applies to terms in $\mathbf{P}_{\mathrm{let}}$ as well.

We now may declare the class of buffer cell objects,[3]

$$\mathbf{letrec}\ \mathit{Cell}(sv) = s\triangleright[\mathsf{read}(r).\ r\triangleleft\mathsf{value}(v), \mathit{Cell}(sv)\ \&\ \mathsf{write}(u).\ \mathit{Cell}(su)]\cdots$$

and instantiate two cells: one holding an integer, the other holding a boolean value,

$$\cdots\ \mathbf{in}\ (\nu t)(\mathit{True}(t), \mathit{Cell}(bt)), (\nu f)(\mathbf{5}(f), \mathit{Cell}(nf))$$

for some term-variables *True* and **5**. But we can do more than this: we can have objects with different methods waiting for replies from **read** requests (as long as the methods include one labeled with **value**), as in,

$$b\triangleleft\mathsf{read}(r), r\triangleright[\mathsf{value}(x).\ P\ \&\ \mathsf{print}(y).\ Q],$$
$$n\triangleleft\mathsf{read}(r'), r'\triangleright[\mathsf{value}(x).\ P'\ \&\ \mathsf{status}(z).\ R]$$

4 Polymorphic Type Assignment

This section introduces an extension of the monomorphic typing assignment system by allowing kinded abstraction and kinded application over sequences of types.

[3] $\mathbf{letrec}\ X(\tilde{x}) = P\ \mathbf{in}\ Q$ is short for $\mathbf{let}\ X(\tilde{x}) = \mathbf{rec}\ X(\tilde{x}).P\ \mathbf{in}\ Q$

Types for Terms. Types for terms fall into two classes, corresponding to (monomorphic) sequences of types for names and polymorphic types constructed using $\forall$. Following Mitchell [7] and the terminology of Martin-Löf's type theory [9], we will call these classes *universes.*

The first universe $\mathbf{U}_1$ is the set of all (finite) sequences of simple types. Greek letters τ, τ' will range over $\mathbf{U}_1$. The second universe $\mathbf{U}_2$ contains all types in the first universe, as well as polymorphic types built from kinded abstraction on type-variables $\forall t : k.\sigma$, where $\sigma, \sigma', \ldots$ range over $\mathbf{U}_2$.

As a consequence of the definition, universal quantifiers may only occur at the top level of types. A polymorphic type of the form $\forall t_1 : k_1. \cdots \forall t_n : k_n.\tau$ will often be written $\forall t_1 : k_1 \cdots t_n : k_n.\tau$ or simply, $\forall \tilde{t} : \tilde{k}.\tau$. We say a $\mathbf{U}_2$ type $\forall \tilde{t} : \tilde{k}.\tau$ is *closed* if $\{\tilde{t}\}$ contains all type variables in τ.

Typings for terms may now be extended to the second universe. We have seen that a typing of the form $\{x_1 : \alpha_1, \ldots x_n : \alpha_n\}$ may be written as $\tilde{x} : \tilde{\alpha}$ (we drop the braces here), by making $\tilde{x} = x_1 \cdots x_n$ and $\tilde{\alpha} = \alpha_1 \cdots \alpha_n$. This allows to abstract the typing on some type-variable t constrained by some kind k, to obtain a typing $\tilde{x} : \forall t : k.\tilde{\alpha}$ of the second universe.

Polymorphic Type Assignment. We will write $K, B \vdash_\forall P \Rightarrow \Gamma$ if statement $P \Rightarrow \Gamma$ is provable from typing K and basis B using the axioms and the rules of the system $\mathrm{TA}_\forall$ below.

The *polymorphic type assignment system* $\mathrm{TA}_\forall$ is defined by the rules in figure 1, together with the following ones,

$$\textsc{Let}\quad \frac{K, B \vdash P \Rightarrow \tilde{x} : \sigma \qquad K, B \cdot X : \sigma \vdash Q \Rightarrow \Gamma}{K, B \vdash \mathbf{let}\, X(\tilde{x}) = P \,\mathbf{in}\, Q \Rightarrow \Gamma}$$

$$\forall\text{-}\textsc{Intro}\quad \frac{K \cdot t : k, B \vdash P \Rightarrow \tilde{x} : \sigma}{K, B \vdash P \Rightarrow \tilde{x} : \forall t : k.\sigma}\ (t \text{ not free in } B)$$

$$\forall\text{-}\textsc{Elim}\quad \frac{K, B \vdash P \Rightarrow \tilde{x} : \forall t : k.\sigma \qquad K \vdash \alpha : k}{K, B \vdash P \Rightarrow \tilde{x} : \sigma[\alpha/t]}$$

where $\sigma[\alpha/t]$ denotes the $\mathbf{U}_2$ type obtained by replacing each variable t in σ by type α. Furthermore, in rule Var, $\tilde{\alpha}$ should be replaced by σ.

All the properties of TA discussed in section 2 easily extend to $\mathrm{TA}_\forall$,[4] in particular the subject-reduction property, which implies that all terms in $\mathbf{P}_{\mathrm{let}}$ that may be assigned a polymorphic types will not run into errors at execution time.

[4]Particular care must be taken with let-processes. In fact, $\mathbf{let}\, X(\tilde{x}) = P \,\mathbf{in}\, Q$ and $Q[P/X]_{\tilde{x}}$ only have the same typings when P occurs in Q, for else, if P is not typable, the expanded process may be typable but the let-process not.

Consider the buffer cell at the end of Section 2. By applying rule $\forall$-Intro twice we obtain the typing,

$$sv : \forall t : (), u : \langle \mathsf{value} : t \rangle.\ [\mathsf{read} : u, \mathsf{write} : t]\ t$$

and then, from basis $B = \{\mathit{True} : \mathit{Bool},\ \mathbf{5} : \mathit{Nat}\}$, for some types *Bool* and *Nat*, we have that the **letrec** term at the end of the previous section has a typing containing,

$$\{b : [\mathsf{read} : [\mathsf{value} : \mathit{Bool}, \mathsf{print} : \alpha], \mathsf{write} : \mathit{Bool}],\ n : [\mathsf{read} : [\mathsf{value} : \mathit{Nat}, \mathsf{status} : \beta], \mathsf{write} : \mathit{Nat}]\}$$

for some types α and β.

5 A Simpler System

In this section we study some properties of calculi $\mathbf{P}$ and $\mathbf{P}_{\mathrm{let}}$ and their typing assignment systems TA and $\mathrm{TA}_\forall$, which will eventually lead to the formulation of a simpler polymorphic typing assignment system, suitable to derive a simple algorithm to extract the principal typing of a process.

System $\mathrm{TA}_\forall$ versus System TA. As it happens in ML, closed terms (possibly containing let-declarations) have polymorphic typings if and only if they have monomorphic typings.

Lemma 5.1 *Let P be a closed term in $\mathbf{P}_{\mathrm{let}}$. There is a $\mathbf{U}_2$ typing for P if and only if P has a $\mathbf{U}_1$ typing.*

Proof: ($\Rightarrow$) We prove by induction on the length of derivations that if $K, B \vdash_\forall P \Rightarrow \tilde{x} : \forall \tilde{t} : \tilde{k}.\tau$ then $K, B \vdash_\forall P \Rightarrow \tilde{x} : \tau$. Rules Msg, Obj, Conc and Rec are only defined for $\mathbf{U}_1$ typings. When the last rule is $\forall$-Intro or $\forall$-Elim, use $\forall$-Elim as many times as needed to obtain a $\mathbf{U}_1$ typing. Since $X(\tilde{v})$ is not a closed term, the remaining interesting case is Let. So assume $K, B \vdash_\forall P \Rightarrow \tilde{x} : \sigma$ and $K, B \cdot X : \sigma \vdash_\forall Q \Rightarrow \Gamma$, for Γ a $\mathbf{U}_2$ typing. The induction hypothesis is that $K, B \vdash_\forall P \Rightarrow \tilde{x} : \tau$ and $K, B \cdot X : \sigma \vdash_\forall Q \Rightarrow \Delta$, for Δ a $\mathbf{U}_1$ typing. The result follows by rule $\forall$-Intro followed by Let.
($\Leftarrow$) A $\mathbf{U}_1$ basis is already a $\mathbf{U}_2$ basis. □

There are however terms in $\mathbf{P}_{\mathrm{let}}$ (and in $\mathbf{P}$) which have typings in $\mathbf{U}_2$ but not in $\mathbf{U}_1$. Examples are terms where a free variable X occurs twice with different types, instances of some type σ, with $X : \sigma$ in the $\mathbf{U}_2$ basis.

Also similarly to what happens in ML, let-terms can be expanded thus completely eliminating the need for let-declarations. In our case, let-term are expanded according to the structural congruence rule $\mathbf{let}\, X(\tilde{x}) = P \,\mathbf{in}\, Q \equiv Q[P/X]_{\tilde{x}}$, yielding terms with the same typings, provided P occurs in Q. The following Lemma follows directly from the fact that structural congruent processes have the same typings (but see footnote 4.)

Lemma 5.2 *Let P be a term in $\mathbf{P}_{\mathrm{let}}$ and let P' be its let-expanded counterpart. Then,*

$$K, B \vdash_{\forall} P \Rightarrow \Gamma \iff K, B \vdash_{\forall} P' \Rightarrow \Gamma$$

Recursion versus Replication. It is well known that explicit recursion can be eliminated in favor of replication (cf. [5].) A replicated object of the form $!P$ is meant to denote an unbounded number of copies of P, captured by the structural congruence rule $!P \equiv P, !P$. A recursive object $\mathbf{rec}\, X(\tilde{v}).P$ can be transformed into a replicated object of the form,

$$(\nu c)(c \triangleleft \mathsf{recur}(\tilde{v}), !c \triangleright [\mathsf{recur}(\tilde{v}).\ P'])$$

where $c \notin \mathcal{FN}(P)$ and P' is obtained from P by replacing free occurrences of $X(\tilde{u})$ by $c \triangleleft \mathsf{recur}(\tilde{u})$.[5]

In this way we can replace terms of the form $\mathbf{rec}\, X(\tilde{v}).P$ by their replicated forms. The corresponding typing assignment system $\mathrm{TA}_{!\forall}$ is obtained by replacing in $\mathrm{TA}_{\forall}$, rule REC by rule REPL below.

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

On the other hand, replication may be eliminated in favor of recursion. For example, a replicated term $!P$ can be transformed into,[6]

$$(\nu c)(c \triangleleft \mathsf{repl},\ \mathbf{rec}\, X(c).\ c \triangleright [\mathsf{repl}.\ P, c \triangleleft \mathsf{repl}, X(c)])$$

It is then easy to see that systems $\mathrm{TA}_{\forall}$ and $\mathrm{TA}_{!\forall}$ are equivalent in the following sense.

Lemma 5.3 *Let P be a term in $\mathbf{P}_{\mathrm{let}}$ and let P' be its replicated counterpart. Then,*

$$K, B \vdash_{\forall} P \Rightarrow \Gamma \iff K, B \vdash_{!\forall} P' \Rightarrow \Gamma$$

Proof: ($\Rightarrow$) Replace in the deduction of $\mathbf{rec}\, X(\tilde{v}).P$ occurrences of $K, B \cdot X : \tilde{\alpha} \ \vdash X(\tilde{u}) \Rightarrow \{\tilde{u} : \tilde{\alpha}\}$ by $K, B \ \vdash c \triangleleft \mathsf{recur}(\tilde{u}) \Rightarrow \{\tilde{u} : \tilde{\alpha}, c : [\mathsf{recur} : \tilde{\alpha}]\}$. Since c is not free in P we have a deduction of $K, B \ \vdash_{!\forall} P' \Rightarrow \Gamma \cdot \tilde{v} : \tilde{\alpha} \cdot c : [\mathsf{recur} : \tilde{\alpha}]$. The result follows by rules OBJ, REPL, CONC and SCOP, in this order.
($\Leftarrow$) Suppose $K, B \ \vdash_{!\forall} !P' \Rightarrow \Gamma$. Then we have a deduction of $K, B \ \vdash P \Rightarrow \Gamma$ and hence one of $K, B \cdot X : [\mathsf{repl} : \epsilon] \ \vdash P \Rightarrow \Gamma$, where ϵ denotes the empty sequence of types. The result follows from the fact that $K, B \ \vdash c \triangleleft \mathsf{repl} \Rightarrow \{c : [\mathsf{repl} : \epsilon]\}$ and $K, B \ \vdash X(c) \Rightarrow \{c : [\mathsf{repl} : \epsilon]\}$, by rules CONC, CONC, OBJ, REC, CONC and SCOP, in this order. □

A Simpler Polymorphic System. Although equivalent to $\mathrm{TA}_{\forall}$ for closed terms, system TA cannot deal with terms were free variables occur with different types, instances of some polymorphic type. For we often want to be able to type open terms, we define a simpler calculus together with a simpler polymorphic typing assignment system equivalent to $\mathrm{TA}_{\forall}$, based on the above results.

Terms of the simplified calculus $\mathbf{P}_!$ are obtained from $\mathbf{P}$ by replacing the constructor $\mathbf{rec}\, X(\tilde{v}).P$ by $!P$. Explicit recursive processes will be compiled into replication through the method described above. Let-declarations will be expanded according to the rule in the structural congruence. The corresponding polymorphic typing assignment system $\mathrm{TA}_{\forall'}$ is obtained from TA by replacing rule REC by rule REPL and by replacing rule VAR by rule VAR$_{\forall}$ below.

$$\textsc{Var}_{\forall} \quad \frac{K \ \vdash \tilde{\alpha} : \tilde{k}}{K, B \cdot X : \forall \tilde{t} : \tilde{k}.\tau \ \vdash X(\tilde{v}) \Rightarrow \tilde{v} : \tau\{\tilde{\alpha}/\tilde{t}\}}$$

Since we don't have rule $\forall$-INTRO anymore we should start deductions in $\mathrm{TA}_{\forall'}$ with closed bases, that is, bases with closed predicates. Then we have the following equivalence result between the polymorphic system $\mathrm{TA}_{\forall}$ and its simplified form $\mathrm{TA}_{\forall'}$.

Theorem 5.4 *Let P be a term in $\mathbf{P}_{\mathrm{let}}$ term and let P' in $\mathbf{P}_!$ be its replicated, let-expanded counterpart. Let B be a closed basis and Γ a $\mathbf{U}_1$ typing. Then,*

$$K, B \vdash_{\forall} P \Rightarrow \Gamma \iff K, B \vdash_{\forall'} P' \Rightarrow \Gamma$$

Proof: ($\Rightarrow$) (Outline) When the last rule is $\forall$-ELIM we follow the derivation backwards until we find a rule other than $\forall$-INTRO or $\forall$-ELIM. If this rule is VAR, the result follows by rule VAR$_{\forall}$ with an adequate choice of $\tilde{\alpha}$. Otherwise the result follows by a simple induction on the structure of deductions, with help from Lemmas 5.3 and 5.2.
($\Leftarrow$) A simple induction on the structure of deductions by using Lemmas 5.3 and 5.2. □

6 Principal Typings and Typing Inference

This section introduces a notion of principal typings, from which all typings that make a process well-typed can be derived. Then we present an algorithm to extract the principal typing of a process, together with a proof of its correctness with respect to the typing assignment system $\mathrm{TA}_{\forall'}$. Although there is an algorithm which incrementally builds a typing for a process

[5] Label recur is, of course, arbitrary.
[6] Term $\mathbf{rec}\, X.(X, P)$ is not a valid term of the calculus.

in $\mathbf{P}_{\text{let}}$, in the style of Damas and Milner [1], the one presented here is much simpler and so is the proof of its correctness.

The algorithm is a simple extension of that in [11], which in turn is based on that of Vasconcelos and Honda [10] for the polyadic π-calculus, and on that of Wand [12] for the λ-calculus.

Principal Kinded Typings.

A ***substitution on types*** is a mapping $s : \mathbf{V} \to \mathbf{T}$ from type variables to types. Such a substitution can be easily extended to types, typings and kinds. Following [8], a ***kinded substitution*** is a pair (K, s) composed of a kinding K and a substitution s. We say a kinded substitution (K', s) ***respects*** a kinding K if and only if $K' \vdash st : sk$ whenever $t : k \in K$.

A kinded substitution (K, s) is *more general* than (K', r) if there is a substitution u such that $r = us$ and (K', u) respects K.

A ***kinded set of equations*** is a pair (K, E) composed of a kinding K and a set of equations of the form $\alpha = \beta$, for α and β types in $\mathbf{T}$. We say a kinded substitution (K, s) is a *unifier* of (K', E) if and only if (K, s) respects K' and $s\alpha = s\beta$, for all $\alpha = \beta \in E$.

Theorem 6.1 (Kinded unification [8]) *There is an algorithm which, given any kinded set of equations, computes a most general unifier if it exists, and reports failure otherwise.*

We say that a kinded typing (K', Δ) is *an instance* of (K, Γ) (or alternatively that (K, Γ) is *more general than* (K', Δ)) if there is a substitution s such that (K', s) respects K and $s\Gamma \subseteq \Delta$. One important fact about instances is that every instance of a well-typing is also a well-typing.

Lemma 6.2 *If $K, B \vdash_\forall P \Rightarrow \Gamma$ and (K', Δ) is an instance of (K, Γ), then $K', B \vdash_\forall P \Rightarrow \Delta$.*

All well-typings for a given process are instances of its ***principal kinded typing***. We say a kinded typing (K, Γ) is ***principal*** for a process P under basis B if and only if,

i. $K, B \vdash_\forall P \Rightarrow \Gamma$, and

ii. if $K', B \vdash_\forall P \Rightarrow \Delta$, then (K', Δ) is an instance of (K, Γ).

It should be obvious that the principal typing of a process, when it exists, is unique up to renaming of type variables, and that it contains exactly the free names in the process.

Theorem 6.3 (Existence of principal typings) *If P is typable then there exists a principal kinded typing for P. It can be effectively computed.*

The Algorithm.

The algorithm builds from a basis B_0 and a process P_0 with recursive terms compiled into replication and let declarations expanded, and with all bound names renamed to be distinct, a typing and a kinded set of equations to be submitted to the kinded unification procedure.

Suppose the algorithm to be described produces a typing Γ_0 and a kinded set of equations (K, E), and use kinded unification on the set of equations. If (K', s) is a unifier of (K, E), then $s\Gamma_0$ is a well-typing for P_0 under kinding K' and basis B_0. Conversely, if P_0 is typable, then all its P-typings under kinding K' and basis B_0 are of the form $s\Gamma_0 \restriction P_0$, for (K', s) a unifier of (K, E).

If Γ is a typing, we will write Γa for the type associated with name a in Γ, and $\Gamma\tilde{a}$ for the sequence of types associated with the sequence of names $\tilde{a}$ in Γ. Similarly, we will write BX for the $\mathbf{U}_2$ type associated with term variable X in basis B.

Input: A basis B_0 and a term P_0 in $\mathbf{P}_!$ with all bound names renamed to be distinct.

Initialization: Set $E = \emptyset$, $G = \{P_0\}$, Γ_0 to a typing assigning to all names in P_0 distinct type-variables, and K to a kinding assigning to all variables in Γ_0 an empty kind $\langle\rangle$.

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

- **Case** P **is** $a \triangleleft l(\tilde{v})$: Generate the equation $\Gamma_0 a = t$ and the kind assignment $t : \langle l : \Gamma_0\tilde{v}\rangle$, for t a fresh variable.
- **Case** P **is** $a \triangleright [l_1(\tilde{x}_1).P_1 \,\&\cdots\&\, l_n(\tilde{x}_n).P_n]$: Generate the equation $\Gamma_0 a = [l_1 : \Gamma_0\tilde{x}_1, \ldots l_n : \Gamma_0\tilde{x}_n]$ and the goals $P_1, \ldots P_n$.
- **Case** P **is** Q, R: Generate the goals Q and R.
- **Case** P **is** $(\nu x)Q$ **or** $!Q$: Generate the goal Q.
- **Case** P **is** $X(\tilde{v})$: Generate the equations $\Gamma_0\tilde{v} = \tau\{\tilde{u}/\tilde{t}\}$ and the kind assignments $\tilde{u} : \tilde{k}$, for $B_0X = \forall\tilde{t} : \tilde{k}.\tau$ and $\tilde{u}$ fresh variables.

To build the principal kinded typing of a term P_0, we use the above algorithm on P_0 and then the kinded unification algorithm on the resulting kinded set of equations (K, E). If (K, E) has no solutions, then P_0 is not typable under basis B_0. Otherwise let (K', s) be the most general unifier of (K, E). Then, $(K', s\Gamma_0 \restriction P_0)$ is the principal typing of P_0. If follows by Lemma 6.2 that every instance of $(K', s\Gamma_0 \restriction P_0)$ is a well kinded typing for P_0.

Correctness of the Algorithm.

Following [10, 12], we prove that the algorithm preserves a certain invariant and terminates. To simplify the statement of the invariant, we introduce some notation. Let (K', s) be a kinded substitution and (K, E) a kinded set of equations. We say (K', s) solves (K, E), denoted $(K', s) \models (K, E)$, if and only if (K', s) is a unifier of (K, E). Given a typing Γ_0 and a basis B_0, we write $(K, s) \models P$ to mean $K, B_0 \vdash_{\forall'} P \Rightarrow s\Gamma_0$, for some process P. If G is a set of goals, we write $(K, s) \models G$ if and only if $(K, s) \models P$ for each process P in G. Finally, we say (K', s) solves (K, E, G), denoted $(K', s) \models (K, E, G)$, if and only if $(K', s) \models (K, E)$ and $(K', s) \models G$. The invariant of the algorithm is as follows.

$$\text{(Soundness)} \quad \forall(K', s).(K', s) \models (K, E, G) \;\Rightarrow\; K', B_0 \vdash_{\forall'} P_0 \Rightarrow s\Gamma_0$$

$$\text{(Completness)} \quad K, B_0 \vdash_{\forall'} P_0 \Rightarrow \Gamma \;\Rightarrow\; \exists(K', s).(K', s) \models (K, E, G) \wedge \Gamma{\restriction}P_0 = s\Gamma_0{\restriction}P_0$$

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

$$\forall(K', s).(K', s) \models (K, E) \;\Rightarrow\; K', B \vdash_{\forall'} P_0 \Rightarrow s\Gamma_0$$

$$K, B \vdash_{\forall'} P_0 \Rightarrow \Gamma \;\Rightarrow\; \exists(K', s).(K', s) \models (K, E) \wedge \Gamma{\restriction}P_0 = s\Gamma_0{\restriction}P_0$$

so that the algorithm only produces well kinded typings for the input process, and if the input process is typable, then its P-well-typings under kinding K' and basis B_0 are given by $s\Gamma_0{\restriction}P_0$, for some unifier (K', s) of the kinded set of equations produced.

Theorem 6.4 (Termination) *The algorithm always terminates.*

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

Theorem 6.5 (Correctness) *The invariants are established by the initialization step and preserved by each case in the loop.*

Proof: (Soundness) The first part is trivial; the second follows by a simple induction on the structure of deductions.
(Completeness) For the first part take s to be the substitution that assigns α to t whenever $x : t \in \Gamma_0{\restriction}P_0$ and $x : \alpha \in \Gamma{\restriction}P_0$, and take K' to be K. Then (K, s) unifies $(K, \emptyset)$; and $K \vdash_k P_0 \succ s\Gamma_0$ implies $K \vdash_k P_0 \succ \Gamma{\restriction}P_0$ by lemma 2.1, which in turn implies $K \vdash_k P_0 \succ \Gamma$ by consecutive applications of rule Weak; and $\Gamma{\restriction}P_0 = s\Gamma_0{\restriction}P_0$.
The proof of the second part follows by induction on the structure of terms. Cases other than $X(\tilde{v})$ are proved in [11].

Case $X(\tilde{v})$. Assume $K, B_0 \vdash_{\forall'} X(\tilde{v}) \Rightarrow \Gamma$. We need to show that $\exists(K', s) : (K', s) \models (K \cup \{\tilde{u} : \tilde{k}\}, \{\Gamma_0\tilde{v} = \tilde{\alpha}\{\tilde{u}/\tilde{t}\}\}) \wedge \Gamma{\restriction}P_0 = s\Gamma_0{\restriction}P_0$. By the typing rules we know that $K \vdash \tilde{\beta} : k$ and $\Gamma{\restriction}P_0 = \{\tilde{v} : \tilde{\alpha}\{\tilde{\beta}/\tilde{t}\}\}$. Make $s = \{\tilde{u} \mapsto \tilde{\beta}, \Gamma\tilde{v} \mapsto \alpha\{\tilde{\beta}/\tilde{t}\}\}$. Then we have that $K \vdash_{\forall'} s\tilde{u} : s\tilde{k} = \tilde{k}$ since neither $\tilde{u}$ nor $\Gamma\tilde{v}$ occur in $\tilde{k}$; and that $s\Gamma_0\tilde{v} = s\alpha\{\tilde{\beta}/\tilde{t}\}$; and also that $\{\tilde{v} : \tilde{\alpha}\{\tilde{\beta}/\tilde{t}\}\} = s\{\tilde{v} : \Gamma\tilde{v}\}$. □

7 Related Work and Further Issues

There is a remarkable parallel between the typing assignment system presented in this paper and that of ML. Two of such similarities were discussed at the beginning of Section 5. Another one concerns let-terms and function application in ML.

There are ML-terms of the form **let** $x = M$ **in** N that can be typed whereas the corresponding function application $(\lambda x.N)M$ cannot. Now, we have no form of application and abstraction over processes, but we have a device by which we can simulate the sharing of a process (c.f. [5]). For example the idea of sharing term P in term Q via the variable X, can be materialized into,

$$(\nu c)(!c \rhd [\mathsf{share}(\tilde{x}).P], Q')$$

where c is a fresh name and Q' is obtained from Q by replacing occurrences of the form $X(\tilde{a})$ by $c \lhd \mathsf{share}(\tilde{a})$. Notice the usage of replication, needed for effective sharing of process P when X occurs more than once in Q. Similarly to ML, **let** $X(\tilde{x}) = P$ **in** Q may be typable, whereas $(\nu c)(!c \rhd [\mathsf{share}(\tilde{x}).P], Q')$ may no be so, for X may occur in Q with different types (instances of a polymorphic type for 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.

In general, typing constraints have a drawback in that there are many meaningful and useful programs that cannot be typed. The present system is no exception. In particular, encodings of recursive data structures such as natural numbers and lists cannot be typed in the present system.

A possible extension to the type assignment system here proposed encompasses adding a type constructor denoting a recursive type and a mapping from types into (possibly infinite) labeled trees. Then, by identifying type equivalence with equality of the associated trees, we could introduce a rule allowing to replace a type in a typing by an equivalent type. Such was the approach taken by Honda and the author [10] for the polyadic π-calculus. The system obtained enjoys all the desirable properties, including the subject reduction and the existence and computability of principal typings.

Conclusion

We presented a basic calculus aiming to capture some essential notions present in systems of concurrent objects communicating via asynchronous message passing, together with a polymorphic typing assignment system for the calculus. Types are assigned to names and are intended to describe the kind of messages an object associated with the name is able to receive. Terms are assigned a collection of name-type pairs called typings, making it possible to abstract the typing on some particular type-variable, thus obtaining a polymorphic typing. A form of let declaration allows to define an object of a polymorphic typing and to use it several times with different types, instances of the declared object.

The typing system assigns a type to each free name in a program, thus specifying in some sense the interface of the program. Programs that conform to the typing discipline were shown not to run into errors. Furthermore, there is an algorithm to derive the principal typing of a program, from which all typings that make the program well-typed can be extracted.

The polymorphic system proposed shows remarkable similarities to ML. In particular, like in ML, the fact that accounts for the extra flexibility of the polymorphic system is not so much let-terms themselves but else the existence of variables of a polymorphic type. Furthermore, we saw that there are let-terms of the form $\textbf{let}\, X(\tilde{x}) = P \,\textbf{in}\, Q$ that can be typed whereas the corresponding term where P is shared in Q via variable X cannot.

The approach seems an interesting basis from which explore further aspects present in objects, namely the notion of inheritance (by introducing new or redefining existing methods in objects) and that of subtyping (by introducing new components in a record type) as well the relationship between these. Also, an extension of the typing system to include recursive types, indispensable to type objects denoting basic data such as natural numbers and lists, can be easily done along the lines of [10].

On the pragmatic side, one should study the applicability of the calculus as a means to describe semantics and types of object-oriented concurrent programming languages such as Actor based languages, Concurrent Smalltalk, ABCL and POOL, as well as a clean incorporation of functions as a particular discipline of object definition and usage.

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] Carl Hewitt. Viewing control structures as patterns of passing messages. *Artificial Inteligence*, 8(3):323–364, 1977.

[3] Kohei Honda. Types for Dyadic Interaction. In *Proceedings of CONCUR'93*, Springer-Verlag, LNCS, August 1993.

[4] Kohei Honda and Mario Tokoro. An Object Calculus for Asynchronous Communication. In *1991 European Conference on Object-Oriented Computing*, pages 141–162, Springer-Verlag, 1991. LNCS 512.

[5] 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.

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

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

[8] Atsushi Ohori. A compilation method for ML-style polymorphic record calculi. In *19th ACM Symposium on Principles of Programming Languages*, pages 154–165, 1992.

[9] P. Martin-Löf. *Intuitionistic Type Theory.* Bibliopolis, Naples, 1974.

[10] Vasco T. Vasconcelos and Kohei Honda. Principal typing-schemes in a polyadic π-calculus. In *Proceedings of CONCUR'93*, Springer-Verlag, LNCS, August 1993.

[11] Vasco T. Vasconcelos and Mario Tokoro. A typing system for a calculus of objects. In *Int. Symp. on Object Technologies for Advanced Software, ISOTAS'93*, Springer-Verlag. LNCS, November 1993. (To appear.).

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