A typing system for a calculus of objects*

Vasco T. Vasconcelos Mario Tokoro†

vasco@mt.cs.keio.ac.jp *mario@mt.cs.keio.ac.jp*

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

Abstract

The present paper introduces an implicitly typed object calculus intended 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, constituting a typing for the process. Programs that comply to the typing discipline are shown not to suffer from runtime errors. Furthermore the calculus possesses a notion of principal typings, from which all typings that make a program well-typed can be extracted. We present an efficient algorithm to extract the principal typing of a process.

1 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 [4], Honda's ν-calculus [3] and Hewitt's actor model [1], we present a basic object-calculus where the notions of objects, asynchronous messages and concurrency are primitive, and introduce a type discipline along the lines of Honda [2] and Vasconcelos and Honda [7] 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 an effectively computable notion of principal typings from which all typings that make a process well-typed can be derived.

*In *1st International Symposium on Object Technologies for Advanced Software*, volume 742 of *LNCS*, pages 460–474. Springer-Verlag, November 1993.

†Also affiliated with Sony Computer Science Laboratory.

Terms of the calculus are built from names by means of a few constructors. Messages of the form $a \lhd l(\tilde{v})$ are directed to an (object located at) name a, select a method labelled with l and carry a sequence of names $\tilde{v}$. Terms of the form $a \rhd [l_1(\tilde{x}_1).P_1 \& \cdots \& l_n(\tilde{x}_n).P_n]$ represent objects located at name a, and comprising a collection of methods, each of which composed of a label l_i, a sequence of formal parameters $\tilde{x}_i$ and an arbitrary process as the method body. Processes are put together by means of the usual concurrent composition. Scope restriction and replication complete the set of constructors the calculus is built from.

Following [2, 7], 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 labelled with l_i carrying a sequence 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. It turns out that the basic typing assignment system possesses no simple notion of principal typing. To provide for a notion of principal typings and to derive an efficient algorithm to extract the principal typing of a process, we use constraints on the substitution of type variables in the form of Ohori's kinds as well as kinded unification [5].

The outline of the paper is as follows. The next section introduces the calculus and section 3 the notion of types and the typing assignment system. Sections 4 and 5 deal with principal typings and typing inference. Section 6 compares the calculus to the polyadic π-calculus [4] and its typing system as proposed in [7]. The last section contains some concluding remarks.

2 The Calculus

This section introduces the calculus to the extent needed for typing considerations. Structural congruence caters for equivalence of terms over concrete syntax and, together with normal forms and message application, makes the formulation of the transition relation quite concise.

2.1 Syntax

Programs are build from names, by means of six basic constructors: messages, objects, concurrent composition, scope restriction, replication and inaction. The set of all terms is $\mathbf{P}$ and $P, Q, \ldots$ will denote particular terms.

Messages are directed to a single object and carry a method selector as well as the message contents itself. *Method selectors* are just labels $l, m, \ldots$ taken from a *set of labels* $\mathbf{L}$. The contents of a message is a sequence of names; that is, messages carry nothing but names. All basic data a program often needs (including for example boolean values and natural numbers) will be coded in such a way that every piece of data is identified by a single name. Names $a, b \ldots$ or $v, x, y, \ldots$ are taken from an infinite *set of names* $\mathbf{N}$. If $x_1, \ldots x_n$ $(n \geq 0)$ are names in $\mathbf{N}$, we write $\tilde{x}$ to mean the sequence $x_1 \cdots x_n$ in $\mathbf{N}^*$. A message targeted to an object identified by name a, selecting a method labelled with l, and carrying a sequence of names $\tilde{v}$, is written as,

$$a \triangleleft l(\tilde{v})$$

Object methods are parameterized by a sequence of names (intended to match the contents of an incoming message) followed by a method body. The body of a method is an arbitrary process. A method with a selector l, a sequence formal parameters $\tilde{x}$, and a body P is written as $l(\tilde{x}).P$. Intuitively, such a method matches a communication $l(\tilde{v})$ and behaves as P with occurrences of names in $\tilde{x}$ replaced by those in $\tilde{v}$.

Objects have a single identifier — a name again — and a collection of methods each labelled with a different label in $\mathbf{L}$. An object with a collection of methods $l_1(\tilde{x}_1).P_1, \ldots l_n(\tilde{x}_n).P_n$ and a name a, is written as,

$$a \triangleright [l_1(\tilde{x}_1).P_1 \,\&\cdots\&\, l_n(\tilde{x}_n).P_n]$$

The remaining constructors in the language are fairly standard in process-calculi. *Concurrent composition* puts together arbitrary processes. If P and Q are two processes, then P, Q denotes the process composed of P and Q running concurrently. *Scope restriction* allows for local name creation avoiding unwanted communications with the exterior. If x is a name and P is a process, then $\nu x P$ denotes the restriction of x to the scope defined by P. Multiple name restrictions on a process $\nu x_1 \cdots \nu x_n P$ will be written $\nu \tilde{x} P$.

Replication accounts for unbounded computation power, and in particular for recursive definition of objects. A replicated object of the form $!\, a \triangleright [l_1(\tilde{x}_1).P_1 \,\&\cdots\&\, l_n(\tilde{x}_n).P_n]$ represents an unbounded number of copies of the object $a \triangleright [l_1(\tilde{x}_1).P_1 \,\&\cdots\&\, l_n(\tilde{x}_n).P_n]$. *Inaction* is the last constructor of the calculus. Denoted by $\mathbf{0}$, it represents the process which does nothing, and could have been defined as $\nu x x \triangleright []$. 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 $\{\tilde{x}\}$. The syntax of the calculus is summarized below.

Definition 2.1 (Syntax) Let $\mathbf{N}$ be an infinite set of *names* and $\mathbf{N}^*$ the set of (finite) sequences over $\mathbf{N}$. Let $\mathbf{L}$ be a set of *labels*. The set of *terms* $\mathbf{P}$ is given by the following grammar.

$$\begin{array}{rcl} P & ::= & a \triangleleft l(\tilde{v}) \quad | \quad a \triangleright [l_1(\tilde{x}_1).P_1 \,\&\cdots\&\, l_n(\tilde{x}_n).P_n] \quad | \quad P, Q \quad | \\ & & \nu x P \quad | \quad !\, a \triangleright [l_1(\tilde{x}_1).P_1 \,\&\cdots\&\, l_n(\tilde{x}_n).P_n] \quad | \quad \mathbf{0} \end{array}$$

where $a, b, \ldots$ and $v, x, y, \ldots$ range over $\mathbf{N}$; $\tilde{v}, \tilde{x}, \tilde{y}, \ldots$ range over $\mathbf{N}^*$; $l, m, \ldots$ range over $\mathbf{L}$ and $P, Q, \ldots$ range over $\mathbf{P}$. Labels $l_1, \ldots l_n$ and names in $\tilde{x}_i$, $i = 1, \ldots n$, are assumed to be pairwise distinct.

For succinctness we will often write $!P$ instead of the more verbose form $!\, a \triangleright [l_1(\tilde{x}_1).P_1 \,\&\cdots\&\, l_n(\tilde{x}_n).P_n]$, but one should keep in mind that we only allow replication over objects.

2.2 Semantics

Objects and scope restriction are the binding operators of the calculus, which justifies the following definition.

Definition 2.2 (Free names) The set of *free names* in P, denoted $\mathcal{FN}(P)$, is inductively defined by:

$$\begin{array}{rcl}
\mathcal{FN}(\mathbf{0}) & = & \emptyset \\
\mathcal{FN}(a \lhd l(\tilde{v})) & = & \{a\} \cup \{\tilde{v}\} \\
\mathcal{FN}(a \rhd [l_1(\tilde{x}_1).P_1 \,\&\cdots\&\, l_n(\tilde{x}_n).P_n]) & = & \{a\} \cup \mathcal{FN}(P_1) \setminus \{\tilde{x}_1\} \cup \cdots \cup \mathcal{FN}(P_n) \setminus \{\tilde{x}_n\} \\
\mathcal{FN}(P, Q) & = & \mathcal{FN}(P) \cup \mathcal{FN}(Q) \\
\mathcal{FN}(\nu x P) & = & \mathcal{FN}(P) \setminus \{x\} \\
\mathcal{FN}(!P) & = & \mathcal{FN}(P)
\end{array}$$

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

Definition 2.3 (Structural congruence) $\equiv$ is the smallest congruence relation over processes defined by the following rules.

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

ii. $(\mathbf{P}, `,', \mathbf{0})$ forms a commutative monoid with equality $\equiv$

iii. $a \rhd [l(\tilde{x}).P \,\&\, m(\tilde{y}).Q] \equiv a \rhd [m(\tilde{y}).Q \,\&\, l(\tilde{x}).P]$

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

v. $!P \equiv P, !P$

There is a *normal form* into which every process can be transformed. Let us call *base-terms* to messages and objects, replicated or not.

Proposition 2.4 (Normal form) *For any process P in $\mathbf{P}$ there is an equivalent process P' of the form,*

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

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

Message application constitutes the basic communication 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.

Definition 2.5 (Message application) Let $l(\tilde{v})$ be the communication of some message, and let $[l_1(\tilde{x}_1).P_1 \,\&\cdots\&\, l_n(\tilde{x}_n).P_n]$ be a collection of methods. Message application is defined by,

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

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.

Definition 2.6 (Reduction) *One-step reduction*, denoted by $\rightarrow$, is the smallest relation generated by the following rules.

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

$$\text{COMM} \qquad \frac{M \bullet C \rightarrow P}{\nu\tilde{x}(\partial, a \triangleleft C, a \triangleright M, \partial') \rightarrow \nu\tilde{x}(\partial, P, \partial')}$$

where ∂ and ∂' represent concurrent composition of base-terms, 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.

Objects are recursive in nature, yet we have no explicit way of providing recursion. Since recursion can be eliminated in favor of replication (see, e.g. [4]), we will freely write,

$$X(\tilde{x}) \stackrel{\text{def}}{=} P \qquad (\{\tilde{x}\} = \mathcal{FN}(P))$$

and let X occur in P, to mean the process,

$$\nu c(c \triangleleft \mathsf{recur}(\tilde{x}), !c \triangleright \mathsf{recur}(\tilde{x}).P') \qquad (c \text{ fresh})$$

where P' is obtained from P by replacing occurrences of $X(\tilde{a})$ for $c \triangleleft \mathsf{recur}(\tilde{a})$.[1] The replicated process can be proved to behave (weakly) similarly to its recursive counterpart. In this way we have an unbounded number of copies of P', each one capable of being activated trough recur with a particular instance of P's "local variables".

Example 2.7 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 buffer is holding. Here is a possible definition.

$$\text{Cell}(sv) \stackrel{\text{def}}{=} s \triangleright [\mathsf{read}(r).\ r \triangleleft \mathsf{value}(v), \text{Cell}(sv) \;\&\; \mathsf{write}(u).\ \text{Cell}(su)]$$

3 Types and Typing Assignment

This section introduces a notion of types for names and a basic typing system to assign types to the free names in terms. We then state some important properties of the typing system.

[1] Label recur is, of course, arbitrary.

3.1 Types for Names

Types are intended to describe some property of the entity they are associated with. In our calculus names identify objects and hence types should represent a property of objects. Objects in **P** do not possess an input-output behavior and thus a function space construct makes no sense. Instead, objects receive messages, messages of a certain form, and that is the property types will describe.

Definition 3.1 (Types) Let $\mathbf{V}$ be an infinite set of *type-variables*. The set of *types* $\mathbf{T}$ is defined by the following grammar.

$$\alpha ::= \quad t \quad | \quad [l_1 : \tilde{\alpha}_1, \ldots l_n : \tilde{\alpha}_n]$$

for $n \geq 0$, where labels $l_1, \ldots l_n \in \mathbf{L}$ are pairwise distinct; $t, t' \ldots$ range over $\mathbf{V}$; $\alpha, \beta \ldots$ over $\mathbf{T}$, and $\tilde{\alpha}, \tilde{\beta} \ldots$ over $\mathbf{T}^*$.

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 labelled with $l_1, \ldots l_n$ and whose arguments of method l_i belong to type $\tilde{\alpha}_i$.

3.2 Typing Assignment

Type assignment formulas are expressions $x : \alpha$, 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.

Definition 3.2 (Typing compatibility) Typings Γ and Δ are compatible, denoted $\Gamma \asymp \Delta$, if and only if,

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

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 $\{x_1 : \alpha_1, \ldots x_n : \alpha_n\}$; $\Gamma \cdot \tilde{x} : \tilde{\alpha}$ denotes typing $\Gamma \cup \{\tilde{x} : \tilde{\alpha}\}$, provided names in $\tilde{x}$ do not occur in Γ; and $\Gamma / \tilde{x}$ denotes the typing Γ with formulas with subjects in $\tilde{x}$ removed.

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

Definition 3.3 (Typing assignment system TA) TA is defined by the following rules.

$$\text{NIL} \qquad \vdash \mathbf{0} \succ \emptyset$$

$$\text{MSG} \qquad \vdash a \triangleleft l(\tilde{v}) \succ \{\tilde{v}:\tilde{\alpha}, a:[l:\tilde{\alpha},\ldots]\}$$

$$\text{OBJ} \quad \frac{\begin{array}{c}(\{a:[l_1:\tilde{\alpha}_1,\ldots l_n:\tilde{\alpha}_n]\} \asymp \Gamma_i, \Gamma_i \asymp \Gamma_j, 1 \le i,j \le n) \\ \vdash P_i \succ \Gamma_i \cdot \tilde{x}_i:\tilde{\alpha}_i\end{array}}{a \triangleright [l_1(\tilde{x}_1).P_1 \,\&\cdots\&\, l_n(\tilde{x}_n).P_n] \succ \{a:[l_1:\tilde{\alpha}_1,\ldots l_n:\tilde{\alpha}_n]\} \cup \Gamma_1 \cup \cdots \Gamma_n}$$

$$\text{SCOP} \quad \frac{\vdash P \succ \Gamma}{\vdash \nu x P \succ \Gamma/x} \qquad \text{CONC} \quad \frac{\vdash P \succ \Gamma \qquad \vdash Q \succ \Delta}{\vdash P, Q \succ \Gamma \cup \Delta} \quad (\Gamma \asymp \Delta)$$

$$\text{REPL} \quad \frac{\vdash P \succ \Gamma}{\vdash !P \succ \Gamma} \qquad \text{WEAK} \quad \frac{\vdash P \succ \Gamma}{\vdash P \succ \Gamma \cdot x:\alpha}$$

Example 3.4 Consider the buffer cell in example 2.7. 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 a message of type $\mathsf{value}:t$, a typing for Cell(sv) is given by,

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

Notice that this is not the only possible well-typing for Cell(sv). In fact, apart from substitutions on variable t, we have that, e.g.

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

is also a well-typing for the process. 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.

3.3 Some Properties of the Typing System

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

Lemma 3.5 *If $\vdash P \succ \Gamma$, then all free names of P occur in Γ and $\vdash P \succ \Gamma\lceil P$.*

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

Lemma 3.6 *If $\vdash P \succ \Gamma$ and $P \equiv Q$, then $\vdash Q \succ \Gamma$.*

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

Theorem 3.7 (Subject Reduction) *If* $\vdash P \succ \Gamma$ *and* $P \twoheadrightarrow Q$, *then* $\vdash Q \succ \Gamma$.

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

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 *run-time error*, and write $P \in \textsc{Err}$, if there exists a term Q such that $P \twoheadrightarrow Q \equiv \nu\tilde{u}(\partial, a \triangleleft l(\tilde{v}), a \triangleright [l_1(\tilde{x}_1).P_1 \& \cdots \& l_n(\tilde{x}_n).P_n], \partial')$ and $[l_1(\tilde{x}_1).P_1 \& \cdots \& l_n(\tilde{x}_n).P_n] \bullet l(\tilde{v})$ is not defined; that is, either $l \notin \{l_1, \ldots l_n\}$ or else $l = l_k \in \{l_1, \ldots l_n\}$ but $len(\tilde{v}) \neq len(\tilde{x}_k)$.

Corollary 3.8 *If* P *is typable, then* $P \notin \textsc{Err}$.

4 Principal typings

We have seen in example 3.4 that the system TA as presented in section 3 possesses no simple notion of principal typings, solely based on substitution of types for type variables, and on the number and nature of labels present in a type. In this section we introduce an alternative presentation of the system, compatible with TA, by using constraints on the substitution of type variables in the style of Ohori [5], which allows to talk about principal typings.

4.1 Kinds and Kinded Typing Assignment

Kinds describe constraints on the substitution of type variables, and are defined as follows.

Definition 4.1 (Kinds) The set of kinds $\mathbf{K}$ is given by all expressions of the form

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

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

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

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. $K, K', \ldots$ 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,

$$\begin{array}{rcccl} K & \vdash & [l_1 : \tilde{\alpha}_1, \ldots l_n : \tilde{\alpha}_n, \ldots] & : & \langle l_1 : \tilde{\alpha}_1, \ldots l_n : \tilde{\alpha}_n \rangle \\ K \cdot t : \langle l_1 : \tilde{\alpha}_1, \ldots l_n : \tilde{\alpha}_n, \ldots \rangle & \vdash & t & : & \langle l_1 : \tilde{\alpha}_1, \ldots l_n : \tilde{\alpha}_n \rangle \end{array}$$

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

Kinded typing assignments are expressions of the form $K \vdash P \succ \Gamma$, where all type variables in typing Γ occur as subjects in kinding K.

We will write $K \vdash_k P \succ \Gamma$ if the statement $P \succ \Gamma$ is provable from kinding K, using the rules and axioms of TA_k below.

Definition 4.2 (Kinded typing assignment system $\mathbf{TA}_k$) TA_k is defined by the rules in TA with sequents of the form $\vdash P \succ \Gamma$ replaced by $K \vdash P \succ \Gamma$ and by replacing rule MSG for rule MSG_k below.

$$\mathrm{MSG}_k \qquad \frac{K \vdash \beta : \langle l : \tilde{\alpha} \rangle}{K \vdash a \lhd l(\tilde{v}) \succ \{\tilde{v} : \tilde{\alpha}, a : \beta\}}$$

We can easily prove that system TA_k is correct with respect to TA. In fact, deductions in TA are valid in TA_k if we start from a kinding assigning an arbitrary kind to every variable appearing in the deduction.

Theorem 4.3 *If $\vdash P \succ \Gamma$ then $K \vdash_k P \succ \Gamma$, for any kinding K binding every variable occurring in the deduction of $\vdash P \succ \Gamma$.*

Proof: By induction on the length of deductions. When the last rule is MSG, make $[l : \tilde{\alpha}, \ldots] = \beta$. Then $K \vdash \beta : \langle l : \alpha \rangle$, and the result follows by rule MSG_k. The remaining cases are trivial. The condition that all variables appearing on the deduction of $\vdash P \succ \Gamma$ are bound in K ensures that all kinded typings in the deduction of $K \vdash_k P \succ \Gamma$ are well-formed. □

Conversely we can prove that, deductions in TA_k can be mapped into deductions in TA, by replacing kinded type variables by types constrained to the kinding.

Theorem 4.4 *If $K \vdash_k P \succ \Gamma$ then $\vdash P \succ \Gamma'$, where Γ' is a typing obtained from Γ by recursively replacing variables t for records of the form $[l_1 : \tilde{\alpha}_1, \ldots l_n : \tilde{\alpha}_n]$ whenever $t : \langle l_1 : \tilde{\alpha}_1, \ldots l_n : \tilde{\alpha}_n \rangle \in K$.*

Proof: By induction on the length of deductions. When the last rule is MSG_k and β is a type variable, since β does not occur in $\tilde{\alpha}$ by the fact that kindings are acyclic, we have $\{\tilde{v} : \tilde{\alpha}\} \asymp \{a : [l : \tilde{\alpha}, \ldots]\}$, and the result follows by rule MSG. The remaining cases are trivial. The fact that kindings are acyclic, ensures that the replacement of type variables for types terminates. □

4.2 Kinded Unification and 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. 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,

$$K' \vdash st : sk \qquad \text{whenever} \qquad t : k \in K$$

A kinded substitution (K, s) is *more general* than (K', r) if there is a substitution u such that,

$$r = us \qquad \text{and} \qquad (K', u) \text{ respects } K$$

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,

$$(K, s) \text{ respects } K' \qquad \text{and} \qquad s\alpha = s\beta \text{ for all } \alpha = \beta \in E$$

Theorem 4.5 (Kinded unification [5]) *There is an algorithm $\mathcal{U}$ which, given any kinded set of equations, computes a most general unifier if it exists, and reports failure otherwise.*

A *kinded typing* is a pair (K, Γ) composed of a kinding K and a typing Γ. 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,

$$(K', s) \text{ respects } K \qquad \text{and} \qquad s\Gamma \subseteq \Delta$$

One important fact about instances is that every instance of a well-typing is also a well-typing.

Lemma 4.6 *If $K \vdash_k P \succ \Gamma$ and (K', Δ) is an instance of (K, Γ), then $K' \vdash_k P \succ \Delta$.*

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

Definition 4.7 (Principal kinded typing) A kinded typing (K, Γ) is *principal* for a process P if and only if,

i. $K \vdash_k P \succ \Gamma$, and

ii. if $K' \vdash_k P \succ \Delta$, 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.

Example 4.8 Recall the buffer cell object of example 2.7, and let t be the type of the value of the cell. By assigning a type variable u (subject to the constraint that it must be assigned a record type having at least a component $\mathsf{value}\!:\!t$) to the object intended to receive the reply to a read request, the principal kinded typing of $\mathrm{Cell}(sv)$ is given by

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

Theorem 4.9 (Existence of principal typings) *If P is typable then there exists a principal kinded typing for P. It can be effectively computed.*

An algorithm to extract the principal kinded typing of a process is described in the next section.

5 Typing Inference

This section introduces an efficient algorithm to extract the principal kinded typing of a process, together with a proof of its correctness with respect to the typing assignment system TA_k. The algorithm is based on that of Vasconcelos and Honda [7] for the polyadic π-calculus, which in turn is based on Wand's [8].

5.1 The Algorithm

The algorithm builds from a process P_0 with all bound names renamed to be distinct, a typing and 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'. Conversely, if P_0 is typable, then all its P-typings under kinding K' 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.

Input: A term P_0 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 $\mathbf{0}$: Generate nothing.

Case P is $a \lhd l(\tilde{v})$: 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 $a \rhd [l_1(\tilde{x}_1).P_1 \,\&\cdots\&\, l_n(\tilde{x}_n).P_n]$: 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.

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. 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 4.6 that every instance of $(K', s\Gamma_0 \restriction P_0)$ is a well kinded typing for P_0.

Example 5.1 To show how the algorithm behaves, let us consider the buffer cell described in Example 2.7. We first eliminate recursion by the method described at the end of Section 2, and rename bound names to obtain the process P_0 below.

$$\nu x(x \lhd \mathsf{recur}(cv), \\ \quad !x \rhd [\mathsf{recur}(c'v').\ c' \rhd [\mathsf{read}(r).\ r \lhd \mathsf{value}(v'), x \lhd \mathsf{recur}(c'v') \,\&\, \mathsf{write}(u).\ x \lhd \mathsf{recur}(c'u)]])$$

Let $\{x : t_x, c : t_c, v : t_v, c' : t'_c, v' : t'_v, r : t_r, u : t_u\}$ be the typing Γ_0. We trace the execution of the algorithm by means of a table. The first column of each line describes the current set of goals. We pick the left-most goal in the list, and write, in the first column of the next line the new set of goals, in the second column the generated equations, in the third column the generated kind assignments, and in the forth column a comment on the rule used. Processes in the set of goals are separated by semi-colons.

$\{P_0\}$			Initial state
$\{x\lhd \mathsf{recur}(cv), !x \rhd[\cdots]\}$			Scope
$\{x\lhd \mathsf{recur}(cv); !x \rhd[\cdots]\}$			Composition
$\{!x \rhd[\mathsf{recur}(c'v').\cdots]\}$	$t_x = t_1$	$t_1 : \langle \mathsf{recur}: t_c t_v\rangle$	Message
$\{x \rhd[\mathsf{recur}(c'v').\cdots]\}$			Replication
$\{c' \rhd[\mathsf{read}(r).\cdots \ \& \ \mathsf{write}(u).\cdots]\}$	$t_x = [\mathsf{recur}: t'_c t'_v]$		Object
$\{r\lhd \mathsf{value}(v'), x\lhd \mathsf{recur}(c'v'); \cdots\}$	$t'_c = [\mathsf{read}: t_r, \mathsf{write}: t_u]$		Object
$\{r\lhd \mathsf{value}(v'); x\lhd \mathsf{recur}(c'v'); \cdots\}$			Composition
$\{x\lhd \mathsf{recur}(c'v'); x\lhd \mathsf{recur}(c'u)\}$	$t_r = t_2$	$t_2 : \langle \mathsf{value}: t'_v\rangle$	Message
$\{x\lhd \mathsf{recur}(c'u)\}$	$t_x = t_3$	$t_3 : \langle \mathsf{recur}: t'_c t'_v\rangle$	Message
$\{\}$	$t_x = t_4$	$t_4 : \langle \mathsf{recur}: t'_c t_u\rangle$	Message

Solving the resulting kinded set of equations using the unification algorithm yields the kinded substitution composed of kinding $K = \{t_v : \langle\rangle, t_r : \langle \mathsf{value} : t_v\rangle\}$ and substitution $s = \{t_x \mapsto [\mathsf{recur}: [\mathsf{read}: t_r, \mathsf{write}: t_v] t_v], t_c \mapsto t'_c \mapsto [\mathsf{read}: t_r, \mathsf{write}: t_v], t_v \mapsto t'_v \mapsto t_u\}$. The principal kinded typing $(K, s\Gamma_0 \lceil P_0)$ for the input process is the one in Example 4.8.

5.2 Correctness and Complexity of the Algorithm

Following [7, 8], we prove that the algorithm preserves a certain invariant and that it 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, we write $(K, s) \models P$ to mean $K \ \vdash_k P \succ s\Gamma_0$, 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.

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

$$\text{(Completness)} \qquad K \ \vdash_k P_0 \succ \Gamma \ \Rightarrow \ \exists (K', s).(K', s) \models (K, E, G) \land \Gamma\lceil P_0 = s\Gamma_0\lceil P_0$$

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

$$\forall (K', s).(K', s) \models (K, E) \ \Rightarrow \ K' \ \vdash_k P_0 \succ s\Gamma_0$$

$$K \ \vdash_k P_0 \succ \Gamma \ \Rightarrow \ \exists (K', s).(K', s) \models (K, E) \land \Gamma\lceil P_0 = s\Gamma_0\lceil P_0$$

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' are given by $s\Gamma_0\lceil P_0$, for some unifier (K', s) of the kinded set of equations produced.

Theorem 5.2 (Termination) *The algorithm always terminates.*

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

Theorem 5.3 (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 $x : t \in \Gamma_0 \lceil P_0$ and $x : \alpha \in \Gamma \lceil P_0$, and take K' to be K. Then (K, s) unifies $(K, \emptyset)$; and $K \vdash_k P_0 \succ s\Gamma_0$ implies $K \vdash_k P_0 \succ \Gamma \lceil P_0$ by lemma 3.5, which in turn implies $K \vdash_k P_0 \succ \Gamma$ by consecutive applications of rule WEAK; and $\Gamma \lceil P_0 = s\Gamma_0 \lceil P_0$.
The proof of the second part follows by induction on the structure of terms. The most difficult cases are when terms are messages and objects; the remaining cases are a simple adaptation of the corresponding in [7].
Case $a \lhd l(\tilde{v})$. Assume $K \vdash_k a \lhd l(\tilde{v}) \succ \Gamma$. We need to show that

$$\exists (K', s).(K', s) \models (K \cup \{t : \langle l : \Gamma_0 \tilde{v} \rangle\}, \{\Gamma_0 a = t\}) \wedge \Gamma \lceil P_0 = s\Gamma_0 \lceil P_0$$

By the typing rules we know that $K \vdash \beta : \langle l : \tilde{\alpha} \rangle$ and $\Gamma \lceil P_0 = \{a : \beta, \tilde{v} : \tilde{\alpha}\}$. Make $s = \{t \mapsto \beta, \Gamma_0 a \mapsto \beta, \Gamma_0 \tilde{v} \mapsto \tilde{\alpha}\}$ and $K' = K$. We then have $K \vdash st : \langle l : s\Gamma_0 \tilde{v} \rangle$; $s\Gamma_0 a = st$ and $\{a : \beta, \tilde{v} : \tilde{\alpha}\} = s\{a : \Gamma_0 a, \tilde{v} : \Gamma_0 \tilde{v}\}$.
Case $a \rhd [l_1(\tilde{x}_1).P_1 \,\&\cdots\&\, l_n(\tilde{x}_n).P_n]$. Assume $K \vdash_k a \rhd [l_1(\tilde{x}_1).P_1 \,\&\cdots\&\, l_n(\tilde{x}_n).P_n] \succ \Gamma$. By induction hypothesis we know that $\exists (K', s).(K', s) \models (K, E, G) \wedge \Gamma \lceil P_0 = s\Gamma_0 \lceil P_0$, and we need to show that,

$$\exists (K'', s').(K'', s') \models (K, E \cup \{\Gamma_0 a = [l_1 : \tilde{\alpha}_1, \ldots l_n : \tilde{\alpha}_n]\}, G \cup \{P_1, \ldots p_n\}) \wedge \Gamma \lceil P_0 = s'\Gamma_0 \lceil P_0$$

By taking $(K'', s') = (K', s)$ we only have to show that $s\Gamma_0 a = s[l_1 : \tilde{\alpha}_1, \ldots l_n : \tilde{\alpha}_n]$ and $K' \vdash_k P_i \succ s\Gamma_0$, for $1 \leq i \leq n$. So let $s\Gamma_0 = \Gamma' \cdot \tilde{x}_1 : \tilde{\alpha}_1 \cdots \tilde{x}_n : \tilde{\alpha}_n$, for some typing Γ' and sequences of types $\tilde{\alpha}_1, \ldots \tilde{\alpha}_n$. By the typing rules we have a derivation of $K \vdash_k P_i \succ \Gamma_i \cdot \tilde{x}_i : \tilde{\alpha}_i$ with $\Gamma_i \asymp \Gamma_j$, $\{a : [l_1 : \tilde{\alpha}_1, \ldots l_n : \tilde{\alpha}_n]\} \asymp \Gamma_i$ and $\Gamma_1 \cup \cdots \Gamma_n = \Gamma'$. Use the WEAK as many times as needed to establish the result, for each i. As for the equations, the result follows from the fact that $\{a : [l_1 : \tilde{\alpha}_1, \ldots l_n : \tilde{\alpha}_n]\} \asymp s\Gamma_0$ and that a occurs in Γ_0. □

As for the complexity, it is easy to see that Γ_0 and the initial kinding can be built in a time linear on P_0, and should take a space proportional to the number of different names in P_0. If types and kinds are implemented with a directed acyclic graph, each equation and kind assignment generated introduces a single node in the graph, and can be computed in a time proportional to the number of names and labels involved. Also, since no copy is needed, each subgoal may be generated in constant time. Hence, the loop executes in a time linear on the size of P_0, produces a number of equations and kind assignments proportional to the number of names in P_0, and introduces a number of new nodes in the graph no larger than the number of names in P_0.

As a corollary, the algorithm to find the principal typing of a process is linear on the size of the textual representation of the input term, provided that the kinded unification algorithm is linear on the number of nodes in the graph.

6 Comparison to the Polyadic π-Calculus

The calculus so far introduced can be easily translated into the polyadic π-calculus [4]. It seems however not be a method of encoding both terms into π-terms and types into

π-types (as presented in [7]) in a satisfactory manner. In what follows, we present two candidates that fail either to transform types or terms.

The first encoding uses summation to represent the disjunctive nature of methods. An object of the form,

$$a \triangleright [l(\tilde{x}).P \;\&\; m(\tilde{y}).Q] \tag{1}$$

is translated into a polyadic π-process of the form,

$$a_l(\tilde{x}).P + a_m(\tilde{y}).Q$$

where a_l and a_m are distinct names. Similarly, a message of the form,

$$a \triangleleft l(\tilde{v}) \tag{2}$$

is translated into a π-process $\overline{a}_l[\tilde{x}].\mathbf{0}$. It is easy to see that terms in our calculus and their images in π-calculus behave strongly similarly. However, instead of translating types, we have to translate typings, for there is not a one-to-one correspondence between the free names in an object and those in its π image. So, for example a typing $\{a : [l : \tilde{t}, m : \tilde{u}]\}$ would have to be translated into the π-typing $\{a_l : (\tilde{\alpha}), a_m : (\tilde{\beta})\}$.

We can also translate terms in our calculus into polyadic π-terms without summation, by using a technique in [3]. The idea is to have a two step protocol, in which a message first asks the object method names, and then selects one of them. Assuming that we have a complete order in $\mathbf{L}$, in such a way that, e.g. $l < m$, the object (1) above can be translated into,

$$\nu lm(a(r).\overline{r}[lm].(l(\tilde{x}).P \mid m(\tilde{y}).Q))$$

In this way, message (2) would be translated into,

$$\nu r(\overline{a}[r].r(lm).\overline{l}[\tilde{v}])$$

Terms in our calculus and their images in the polyadic π-calculus now behave only weakly similarly. The drawback of this encoding is that it assumes we know all the methods in the target object, and therefore it is not context independent. On the other hand types may be translated directly since we now have a one-to-one correspondence between the free names in an object and those in its image. For example, the type $[l : \tilde{t}, m : \tilde{u}]$ could be naturally translated into the π-type $((\,(\tilde{\alpha}), (\tilde{\beta})\,))$.

7 Concluding Remarks

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 typing system for the calculus. Types are assigned to names and are intended to describe the kind of messages objects associated with the name are able to receive. Processes are not assigned types, but else a collection of name-type pairs. The typing system presented assigns a type to each free name in a process, thus specifying in some sense the interface of the process. Programs that conform to the typing discipline were shown not to run into errors. Furthermore, we presented an algorithm to extract the principal typing of a program, from which all typings that make the program well-typed can be extracted. Finally, we discussed the lack of a satisfactory encoding of the present calculus into the polyadic π-calculus.

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 [7]. Another related line of investigation, in the style of [6], encompasses the introduction of variables over process accompanied by a notion of predicative polymorphism, and a ML-like let construct. Such a system would allow the declaration of an object of a polymorphic type, which could be used in a program multiple times with different types, instances of the type of the declared object.

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.

Acknowledgements. The authors wish to thank Kohei Honda for long and fruitful discussions on the nature of concurrency and types for concurrency.

References

[1] Carl Hewitt. A universal, modular actor formalism for artificial intelligence. In *Proceedings of international joint conference on artificial intelligence*, 1973.

[2] Kohei Honda. Types for Dyadic Interaction. In E. Best, editor, *Proceedings of CONCUR'93*, pages 509–523, Springer-Verlag, August 1993. LNCS 715.

[3] 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.

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

[5] Atsushi Ohori. A Compilation Method for ML-style Polymorphic Record Calculi. In *19th ACM Symposium on Principles of Programming Languages*, pages 154–165, 1992.

[6] Vasco T. Vasconcelos. Predicative Polymorphism in π-calculus. May 1993. Keio University.

[7] Vasco T. Vasconcelos and Kohei Honda. Principal typing-schemes in a polyadic π-calculus. In E. Best, editor, *Proceedings of CONCUR'93*, pages 524–538, Springer-Verlag, August 1993. LNCS 715.

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