Recursive Types in a Calculus of Objects *

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

Abstract

We introduce a name-passing calculus featuring objects as guarded labelled-sums, each summand representing a method, and asynchronous labelled messages selecting a branch in the sum. A decidable type assignment system allows to statically verify whether all possible communications in a given program are secure, in the precise sense that no object will ever receive a message for which it does not have an appropriate method. Then we present a recursive type system based on that of Cardone and Coppo for the λ-calculus, and of Vasconcelos and Honda for the polyadic π-calculus. The new system extends the class of typable terms while preserving basic syntactical properties of the simple type system, including subject-reduction and existence and computability of principal typings.

Introduction

Object technologies are a promising paradigm towards the management of complex, concurrent, distributed and open-ended systems. Types are an acknowledged tool to describe protocols of communication, and to isolate classes of programs guaranteed not to suffer from protocol errors at runtime. The calculus of objects brings these two worlds together in an elegant implicitly-typed name-passing calculus.

Inspired in Milner's polyadic π-calculus [8], Honda's ν-calculus [6, 7] and Hewitt's actor model [5], we developed a calculus intended to capture basic features present in most notions of objects, and to give precise (operational) semantics and type inference systems to object-oriented concurrent programming languages [14]. The calculus is built along the trends of name-passing calculi, by introducing labelled-sums representing methods, and asynchronous labelled messages selecting a branch in a sum. A type system assigns types to the free names in (untyped) terms, specifying in some sense the communication protocol of the term, and ensuring that objects in well-typed programs do not receive messages for which they do not possess an appropriate method.

The present paper consolidates the presentation of the calculus of objects [14], and introduces an extension to the basic type system to accommodate recursive types. Recursive types, of the form $\mu t.\alpha$ for t a type-variable and α an arbitrary type, are interpreted

*In Transactions of Information Processing Society of Japan, 35(9):1828–1836, September 1994.

as (possibly infinite) trees; a new type inference rule allows to replace a type by another representing the same tree. This small extension effectively enlarges the class of typable terms while preserving basic syntactic properties including subject-reduction.

Notions of typing subsumption and principal typings are achieved via constraints on the types that may substitute a given type-variable, in the form of Ohori's kinds [10]. Kindings, assigning kinds to type-variables, are required to be acyclic in the simple type system. Recursive kinds (implicitly) arise as cycles in kindings, thus making it unnecessary to introduce new constructors to describe recursive kinds. The existence and computability of principal typings is ensured by an algorithm capable of unifying kinded recursive types in the form of kinded infinite trees.

The outline of the paper is as follows. Section 1 introduces the calculus of objects, Section 2 presents the simple typing assignment system, and Section 3 studies the notion of principal typings. Then, Section 4 introduces the notion of recursive types and the recursive typing assignment system, and Section 5 deals with principal typings in the recursive setting. The last section summarizes the results and points out directions for further work.

1 The calculus of objects

In this section we introduce the calculus of objects, its syntax and semantics. The exposition is brief since the behaviour of programs is not the main topic of the paper.

Syntax. *Terms* are built from an infinite set $\mathbf{N}$ of *names* and a set $\mathbf{L}$ of *labels*, by means of the five constructors below. We use $a, b, \ldots$ and also $v, x, \ldots$ to range over $\mathbf{N}$, and $l, l', \ldots$ to range over $\mathbf{L}$; finite sequences of names are denoted $\tilde{v}, \tilde{x}, \ldots$, with ε representing the empty sequence.

Definition 1.1 (Syntax) The set $\mathbf{P}$ of *terms* is inductively defined by the grammar

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

for $n \geq 0$, where $P, Q, \ldots$ range over $\mathbf{P}$. Names in each $\tilde{x}_i$ for $i = 1, \ldots, n$, as well as labels $l_1, \ldots, l_n$, are assumed to be pairwise distinct.

Terms of the form $a \triangleright [l_1 : (\tilde{x}_1) P_1 \& \cdots \& l_n : (\tilde{x}_n) P_n]$ are intended to describe *objects* located at name a, and possessing a (unordered) collection of methods, each labelled with a distinct label l_i. For each i, the sequence of names $\tilde{x}_i$ represents the formal parameters of the method, whose body is an arbitrary term P_i. *Messages* are terms of the from $a \triangleleft l : \tilde{v}$ and select a method labelled with l in some object located at name a. Names in $\tilde{v}$ constitute the actual contents of the message. The interaction between a message $a \triangleleft l : \tilde{v}$ and an object $a \triangleright [l_1 : (\tilde{x}_1) P_1 \& \cdots \& l_n : (\tilde{x}_n) P_n]$ is the term P_i with names in $\tilde{x}_i$ replaced by those in $\tilde{v}$, provided that l labels some method in the object, and that the lengths of $\tilde{x}_i$ and $\tilde{v}$ match.

Concurrent composition P, Q denotes the term composed of terms P and Q running in parallel. Terms of the from $\nu x P$ allow to create a new name x and use it in a scope restricted to term P. *Replication* provides for unbounded computation power. A term of the form $!P$ intuitively represents as many copies of P as needed, running in parallel.

Semantics. Methods $l:(\tilde{x})P$, and scope restriction νxP, are the binding operators of the calculus, binding free occurrences of $\tilde{x}$, and x, in the respective bodies P. The set of *free names* in a term is defined accordingly. When $\tilde{x}$ and $\tilde{z}$ are two sequences of names of the same length, and the names in $\tilde{x}$ are pairwise distinct, $P\{\tilde{z}/\tilde{x}\}$ denotes the *simultaneous substitution* of the free occurrences of names in $\tilde{x}$ by those in $\tilde{z}$.

Equivalence of terms over concrete syntax is captured by the *structural congruence* relation $\equiv$ (see [14] for a definition). *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, and the launching of its body with the formal parameters replaced by the message contents.

Definition 1.2 (Message application) Let $C = l_i:\tilde{v}$ be the communication of some message, and let $M = [l_1 : (\tilde{x}_1)P_1\& \cdots \& l_n : (\tilde{x}_n)P_n]$ be a collection of methods. The application of C to M, denoted by $M \bullet C$, is the term $P_i\{\tilde{v}/\tilde{x}_i\}$, whenever $1 \leq i \leq n$ and the lengths of $\tilde{v}$ and $\tilde{x}_i$ match.

The following definition of reduction relies on fact that each term in $\mathbf{P}$ can be transformed into a structural congruent term of the form $\nu\tilde{x}\tilde{P}$, where $\tilde{P}$ denotes concurrent composition of messages and objects (replicated or not).

Definition 1.3 (Reduction) *One-step reduction*, denoted by $P \to Q$, is the smallest relation generated by the following rules.

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

$$\textsc{Comm}\quad \nu\tilde{x}(a \triangleright M, a \triangleleft C, \tilde{Q}) \to \nu\tilde{x}(M \bullet C, \tilde{Q})$$

2 Simple typing assignment

In this section we introduce the notion of (simple) types for names and an inference system assigning typings (that is, sets of name-type pairs) to terms. We briefly review some of the properties of the simple typing assignment system.

Types. Types are built from an infinite set $\mathbf{V}$ of *type-variables* and the set $\mathbf{L}$ of labels introduced in the previous section, by means of a single constructor.

Definition 2.1 (Simple Types) The set $\mathbf{T}$ of *types* is inductively defined by the grammar

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

where $l_1, \ldots, l_n$ are pairwise distinct labels. We use $t, t', \ldots$ to range over $\mathbf{V}$; $\alpha, \beta, \ldots$ to range over $\mathbf{T}$; and $\tilde{\alpha}, \tilde{\beta}, \ldots$ to range over sequences of types in $\mathbf{T}^*$.

A record 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 types $\tilde{\alpha}_i$. Similarly to methods in terms, records are unordered, so that $[l : \tilde{\alpha}, l' : \tilde{\beta}]$ and $[l' : \tilde{\beta}, l : \tilde{\alpha}]$ represent the same type.

Simple typing assignment. *Type assignments* are formulas $x\colon \alpha$, for x a name and α a type, where x is called the *subject* and α the *predicate* of the assignment. While we assign types to names, to terms we assign typings. *Typings*, denoted by $\Gamma, \Delta, \ldots$, are sets of type assignments of the form $\{x_1\colon \alpha_1, \ldots, x_n\colon \alpha_n\}$, where no two assignments have the same name as subject. If $\tilde{x} = x_1 \ldots x_n$ is a sequence of distinct names, and $\tilde{\alpha} = \alpha_1 \ldots \alpha_n$ a sequence of types, we often write $\tilde{x}\colon \tilde{\alpha}$ for the typing $\{x_1\colon \alpha_1, \ldots, x_n\colon \alpha_n\}$, and $\Gamma \cdot \tilde{x}\colon \tilde{\alpha}$ for $\Gamma \cup \tilde{x}\colon \tilde{\alpha}$, provided that names in $\tilde{x}$ are not subjects in Γ.

Definition 2.2 (Typing compatibility) Typings Γ and Δ are *compatible*, denoted by $\Gamma \asymp \Delta$, if $\alpha = \beta$ for all $x\colon \alpha \in \Gamma$ and $x\colon \beta \in \Delta$.

Typing assignment statements are formulas $P \triangleright \Gamma$, for any term P and typing Γ. We write $\vdash_{[]} P \triangleright \Gamma$ if the statement $P \triangleright \Gamma$ is provable using the axioms and the rules of system $\mathrm{TA}_{[]}$ below. Whenever $\vdash_{[]} P \triangleright \Gamma$, we say P is *typable*, and call Γ a *well-typing* for P.

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

$$\textsc{Msg} \quad a \lhd l_1 : \tilde{v} \triangleright \{\tilde{v}\colon \tilde{\alpha}_1, a\colon [l_1\colon \tilde{\alpha}_1, \ldots, l_n\colon \tilde{\alpha}_n]\} \quad (n \geq 1)$$

$$\textsc{Obj} \quad \frac{P_1 \triangleright \Gamma_1 \cdot \tilde{x}_1\colon \tilde{\alpha}_1 \qquad \cdots \qquad P_n \triangleright \Gamma_n \cdot \tilde{x}_n\colon \tilde{\alpha}_n}{a \rhd [l_1 : (\tilde{x}_1) P_1 \& \cdots \& l_n : (\tilde{x}_n) P_n] \triangleright \{a\colon [l_1\colon \tilde{\alpha}_1, \ldots, l_n\colon \tilde{\alpha}_n]\} \cup \Gamma_1 \cup \cdots \cup \Gamma_n} \; (\{a\colon [l_1\colon \tilde{\alpha}_1, \ldots, l_n\colon \tilde{\alpha}_n]\} \asymp \Gamma_i, \Gamma_i \asymp \Gamma_j, 1 \leq i,j \leq n, n \geq 0)$$

$$\textsc{Scop} \quad \frac{P \triangleright \Gamma \cdot x\colon \alpha}{\nu x P \triangleright \Gamma} \qquad \textsc{Comp} \quad \frac{P \triangleright \Gamma \quad Q \triangleright \Delta}{P, Q \triangleright \Gamma \cup \Delta} \quad (\Gamma \asymp \Delta)$$

$$\textsc{Repl} \quad \frac{P \triangleright \Gamma}{!P \triangleright \Gamma} \qquad \textsc{Weak} \quad \frac{P \triangleright \Gamma}{P \triangleright \Gamma \cdot x\colon \alpha}$$

We conclude the section with a brief overview of some important properties of $\mathrm{TA}_{[]}$. The free names in a term constitute the interaction points of the term, and the types of these names describe, in some sense, the interface to the term. Well-typings assign types to all free names in terms and, whenever a term is typable, there is a derivation assigning types to exactly the free names in the term.

Subject-reduction ensures that a typing for a term does not change as the term is reduced. As a corollary, typable terms do not run into runtime errors. We say a term P contains a possible *runtime error* if it can be reduced to a term of the form $\nu\tilde{x}(a \rhd M, a \lhd C, \tilde{Q})$ and the message application $M \bullet C$ is not defined. Finally, the property of decidability of typing inference and computability of typings is left to the next section.

3 Principal typings

We would like our type system to verify the two properties below (cf. [1]).

i. *Decidability of the typing inference problem* $\vdash P \triangleright \Gamma?$: given a term P and a typing Γ, decide whether $\vdash P \triangleright \Gamma$.

ii. *Computability of the typing existence problem* $\vdash P \rhd ?$: given a term P, decide whether there is a typing Γ (computable from P) such that $\vdash P \rhd \Gamma$.

The solution to both problems is usually obtained via the notions of subsumption and principal typings (cf. [9]). In general, a typing Γ *subsumes* Γ' if every term with typing Γ also has typing Γ'; a *principal typing* for a term P is a well-typing for P that subsumes all other typings for P. If a type system has a computable notion of principal typings and subsumption is decidable, then the decidability of typing inference, $\vdash P \rhd \Gamma?$, reduces to compute the principal typing for P and verifying if it subsumes Γ. Computability of typing existence, $\vdash P \rhd ?$, amounts to compute the principal typing for P, since this is a well-typing for P.

In λ-calculus, subsumption is often connected with substitution of type-variables for types. When records are a form of types, subsumption also takes into account the number and nature of labels in records. Unfortunately, system $\mathrm{TA}_{[]}$ possess no simple notion of subsumption. The reason seems to lie in that we cannot assign "closed" records to names that never occur at object locations, for we cannot know *all* sorts of messages objects associated with those names may receive (see [14] for a concrete example). Nevertheless, subsumption and principal typings may be recovered with a notion of constraints on the types that may substitute a given type-variable, by using Ohori's kinds [10].

Kinds and kind assignment. Intuitively, kinds describe constraints on the substitution of type-variables: a kind of the form $\langle l_1 \colon \tilde{\alpha}_1, \ldots, l_n \colon \tilde{\alpha}_n\rangle$ denotes the subset of record types containing *at least* the components $l_1 \colon \tilde{\alpha}_1, \ldots, l_n \colon \tilde{\alpha}_n$. In particular, unconstrained (completely free) type-variables have the empty kind $\langle\rangle$.

Definition 3.1 (Kinds) The set $\mathbf{K}$ of *kinds* is given by all expressions of the form

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

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 $n \geq 0$. We use $k, k', \ldots$ to range over $\mathbf{K}$.

Kind assignments are formulas $t \colon k$, for t a type-variable and k a kind. *Kindings*, denoted by $K, K', \ldots$, are sets of kind assignments where no two assignments have the same type-variable as subject. We denote by $\mathrm{dom}\, K$ the set of subjects in K. A kinding is *cyclic* if it contains kind assignments $t_1 \colon k_1, \ldots, t_n \colon k_n$ such that t_{i+1} occurs in k_i, and t_1 occurs in k_n, for $i = 1, \ldots, n$ and $n \geq 1$. Acyclic kindings ensure that the replacement of type-variables by types yields finite types.

Definition 3.2 (Kind assignment to types) A *type α has a kind k under a kinding K*, if $\alpha \rhd k$ can be deduced from K by one of the following axioms. In this case we write $K \vdash_{\langle\rangle} \alpha \rhd k$.

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

Kinded typing assignment. *Kinded typing assignments* are formulas $K \vdash P \rhd \Gamma$, for any acyclic kinding K, term P, and typing Γ. We write $K \vdash_k P \rhd \Gamma$ if the statement $P \rhd \Gamma$ is provable from kinding K using the axioms and rules of system TA_k below. When K is the empty kinding, we simply write $\vdash_k P \rhd \Gamma$.

Definition 3.3 (Kinded typing assignment system TA_k) TA_k is defined by the axioms in the Definition 3.2 of kind assignment to types, by the rules in the Definition 2.3 of $\mathrm{TA}_{[]}$ with formulas $P \triangleright \Gamma$ replaced by $K \vdash P \triangleright \Gamma$, and by the Msg_k-rule below.

$$\mathrm{Msg}_k \quad \frac{K \vdash_{\langle\rangle} \beta \triangleright \langle l\colon \tilde{\alpha}\rangle}{K \vdash a \lhd l\colon \tilde{v} \triangleright \{\tilde{v}\colon \tilde{\alpha}, a\colon \beta\}}$$

The following result shows that TA_k is correct with respect to $\mathrm{TA}_{[]}$.

Theorem 3.4 *Terms in* $\mathbf{P}$ *are typable in* TA_k *if and only they are if typable in* $TA_{[]}$.

Proof: ($\Rightarrow$) Suppose that $K \vdash_k P \triangleright \Gamma$. We build a substitution s from K, and show that $\vdash_{[]} P \triangleright s\Gamma$. Without loss of generality, let $t_1, \dots, t_n$ be the subjects of K. Define a system of equations S with unknowns $t_1, \dots, t_n$, and equations of the form $t_i = [l_{i_1}\colon \tilde{\alpha}_{i_1}, \dots, l_{i_k}\colon \tilde{\alpha}_{i_k}]$ for each kind assignment $t_i\colon \langle l_{i_1}\colon \tilde{\alpha}_{i_1}, \dots, l_{i_k}\colon \tilde{\alpha}_{i_k}\rangle$ in K. Since K is acyclic, standard arguments yield that S has a unique solution in $\mathbf{T}^n$. Let $(\beta_1, \dots, \beta_n)$ be the solution of S, and define a substitution $s\colon \mathbf{V} \to \mathbf{T}$ by putting $st_i = \beta_i$ if $1 \le i \le n$, and $st = t$ otherwise.

Noticing that typing compatibility is preserved by substitution, replace throughout the deduction of $K \vdash_k P \triangleright \Gamma$, occurrences of the Msg_k-rule, formulas, and side-conditions below on the left by the Msg-axiom, formulas, and side-conditions on the right, to obtain a deduction of $\vdash_{[]} P \triangleright s\Gamma$.

$$\frac{K \vdash [l_1\colon \tilde{\alpha}_1, \dots, l_n\colon \tilde{\alpha}_n] \triangleright \langle l_1\colon \tilde{\alpha}_1\rangle}{K \vdash a \lhd l_1\colon \tilde{v} \triangleright \{a\colon [l_1\colon \tilde{\alpha}_1, \dots, l_n\colon \tilde{\alpha}_n], \tilde{v}\colon \tilde{\alpha}_1\}} \qquad a \lhd l_1\colon \tilde{v} \triangleright \{a\colon s[l_1\colon \tilde{\alpha}_1, \dots, l_n\colon \tilde{\alpha}_n], \tilde{v}\colon s\tilde{\alpha}_1\}$$

$$\frac{K \vdash t \triangleright \langle l_1\colon \tilde{\alpha}_1\rangle}{K \vdash a \lhd l_1\colon \tilde{v} \triangleright \{a\colon t, \tilde{v}\colon \tilde{\alpha}_1\}} \qquad a \lhd l_1\colon \tilde{v} \triangleright \{a\colon st, \tilde{v}\colon s\tilde{\alpha}_1\}$$

$$K \vdash P \triangleright \Gamma \qquad P \triangleright s\Gamma$$

$$(\Gamma \asymp \Delta) \qquad (s\Gamma \asymp s\Delta)$$

($\Leftarrow$) Suppose that $\vdash_{[]} P \triangleright \Gamma$. Replace, throughout the deduction of $\vdash_{[]} P \triangleright \Gamma$, occurrences of the Msg-axiom below on the left by the deduction on the right, to obtain a deduction of $\vdash_k P \triangleright \Gamma$.

$$a \lhd l_1\colon \tilde{v} \triangleright \{a\colon [l_1\colon \tilde{\alpha}_1, \dots, l_n\colon \tilde{\alpha}_n], \tilde{v}\colon \tilde{\alpha}_1\} \qquad \frac{[l_1\colon \tilde{\alpha}_1, \dots, l_n\colon \tilde{\alpha}_n] \triangleright \langle l_1\colon \tilde{\alpha}_1\rangle}{a \lhd l_1\colon \tilde{v} \triangleright \{a\colon [l_1\colon \tilde{\alpha}_1, \dots, l_n\colon \tilde{\alpha}_n], \tilde{v}\colon \tilde{\alpha}_1\}}$$

$\square$

Subsumption 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. Following Ohori [10], we say a *kinded substitution* is a pair (K, s) composed of a kinding K and a substitution s. A kinded substitution (K', s) *respects* a kinding K if $K' \vdash_{\langle\rangle} st \triangleright sk$, for all $t\colon k \in K$. A *kinded typing* is a pair (K, Γ) composed of a kinding K and a typing Γ. We say (K, Γ) *subsumes* (K', Δ) if $\mathrm{dom}\, K \subseteq \mathrm{dom}\, K'$, and there is a substitution s such that (K', s) respects K and $s\Gamma \subseteq \Delta$.

Lemma 3.5 *If* $K \vdash_k P \triangleright \Gamma$ *and* (K, Γ) *subsumes* (K', Δ), *then* $K' \vdash_k P \triangleright \Delta$.

Definition 3.6 (PRINCIPAL TYPINGS) A kinded typing (K, Γ) is principal for P if

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

ii. if $K' \vdash_k P \triangleright \Delta$, then (K, Γ) subsumes (K, Δ).

It easy to see that principal typings, when they exist are unique up to renaming of type-variables, and contain exactly the free names in the process. The following result if from [14].

Theorem 3.7 (Existence and computability of principal typings) *If P is typable in TA_k, there exists a principal kinded typing for P. It can be effectively computed.*

Typing inference. Computability of principal kinded typings is ensured by the existence of a unification algorithm that computes the most general unifier of a kinded set of equations [10].

A *kinded set of equations* is a pair (K, E) composed of a kinding K, and of a set of equations E 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 (K', s) respects K and $s\alpha = s\beta$, for all $\alpha = \beta \in E$. 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.

We proposed an efficient algorithm to extract the principal kinded typing of a given term [14]. The algorithm follows the construction tree of the term, annotating constraints on names (in the form of type equations) and on type-variables (in the form of kind assignments), thus producing a kinded set of equations. This kinded set of equations is then submitted to a kinded unification algorithm. The inference algorithm is based on that of Wand for the simply typed λ-calculus [15], and that of Vasconcelos and Honda for the polyadic π-calculus [13].

4 Recursive types and recursive typing assignment

There are numerous meaningful and useful terms that cannot be typed in the simple type system described so far. Terms representing natural numbers, lists and trees are examples of a more general class of terms representing recursive data-structures. Lists, for example, can be built from two basic objects, Nil and Cons, capable of receiving one single *val* message. To such a message, Nil objects reply *nil* (short for $\mathit{nil}\colon\varepsilon$), stating "I am a Nil object," and Cons objects reply $\mathit{cons}\colon fs$, stating "I am a Cons object, and here is my first and second elements."

$$\begin{aligned}\mathrm{Nil}(l) &\stackrel{\text{def}}{=} l \triangleright [\mathit{val}\colon(r)\ r \triangleleft \mathit{nil}\,] \\ \mathrm{Cons}(lfs) &\stackrel{\text{def}}{=} l \triangleright [\mathit{val}\colon(r)\ r \triangleleft \mathit{cons}\colon fs]\end{aligned}$$

Actual lists can be built from these two objects by appropriately hiding the links between the Cons and the Nil cells involved. So, for example, a list of two elements a and b, located at name l, can be written as the term $\nu s(\mathrm{Cons}(l\,a\,s), \nu s'(\mathrm{Cons}(s\,b\,s'), \mathrm{Nil}(s')))$.

Conventional usage of lists forces the type of a Cons cell and that of its second element to coincide. Given a term Cons(lfs), it is easy to see that no typing can assign the same (finite) type to names f and s, for we would need a type α such that $\alpha = [\mathit{val}\colon [\mathit{cons}\colon \beta\alpha, \ldots]]$, for some type β.

Recursive types. We can use equations between type expressions whose solutions are infinite trees. For example, to an equation of the form $\alpha = [\mathit{val}\colon [\mathit{cons}\colon t\alpha, \mathit{nil}\colon \varepsilon]]$ corresponds the tree depicted in Figure 1.

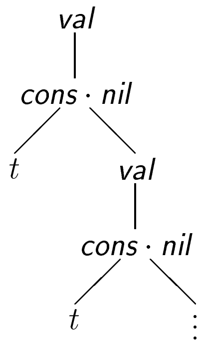

Figure 1: The infinite tree associated with the equation $\alpha = [\mathit{val}\colon [\mathit{cons}\colon t\alpha, \mathit{nil}\colon \varepsilon]]$.

Then we introduce a special notation, $\mu t.\alpha$, to denote the solution of the equation $t = \alpha$, where t is a type-variable and α an arbitrary type. Intuitively, a type $\mu t.\alpha$ may be considered as a finite notation for the possibly infinite tree that solves the equation $t = \alpha$.

Definition 4.1 (Recursive types) The set of *recursive types* $\mathbf{T}_\mu$ is defined by adding to the syntax of simple types in Definition 2.1, a production $\mu t.\alpha$, for any type-variable t in $\mathbf{V}$ and any type α in $\mathbf{T}_\mu$.

Recursive types as infinite trees. Assume that the set $\mathbf{L}$ of labels is totally ordered and equipped with a ranking function rank : $\mathbf{L} \to \mathcal{N}$, where $\mathcal{N}$ is the set of natural numbers. We build the set of *tree labels*—a ranked alphabet $\mathbf{F}$—as the set of strings of pairwise distinct labels in $\mathbf{L}$, such that $l_1 \ldots l_n \in \mathbf{F}$ only if $l_1 \leq \cdots \leq l_n$, and define the rank of a tree-label $l_1 \ldots l_n$ as $\sum_{i=1}^{n} \mathrm{rank}(l_i)$, for $n \geq 0$. A type-variable in $\mathbf{V}$ is a symbol of rank 0.

Following Courcelle [4], a *tree over a ranked alphabet* $\mathbf{F} \cup \mathbf{V}$ is defined as a partial mapping, $\rho : \mathcal{N}_+^* \to \mathbf{F} \cup \mathbf{V}$, from the set of strings over positive integers $\mathcal{N}_+$ into tree-labels in $\mathbf{F}$ or type-variables in $\mathbf{V}$, satisfying a tree domain condition.

Strings of positive integers describe paths in a tree; the nodes of the tree being either type-variables or strings of pairwise distinct labels representing records. If $\rho_1, \ldots, \rho_n$ are n trees, $n \geq 0$, and the rank of the tree-label $f \in \mathbf{F}$ is n, we write $f(\rho_1, \ldots, \rho_n)$ for the tree ρ such that

$$\begin{array}{rcll} \rho(\varepsilon) & = & f & \text{for } \varepsilon \text{ the empty string of positive integers, and} \\ \rho(i\tilde{\jmath}) & = & \rho_i(\tilde{\jmath}) & \text{for } i = 1, \ldots, n \text{ and } \tilde{\jmath} \in \mathcal{N}_+^*. \end{array}$$

Let $\mathbf{T}_\infty$ be the set of all trees over $\mathbf{F} \cup \mathbf{V}$. Introducing a new label $\perp$, we can define a partial ordering of $\mathbf{T}_\infty$ by putting

i. $\bot \sqsubseteq \rho$ for all trees ρ, and

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

It is easy to see that $\mathbf{T}_\infty$ is a c.p.o. and thus any infinite tree can be seen as the least upper bound of a denumerable sequence of finite approximations. For example, the infinite tree associated with equation $\alpha = [\mathit{val}\colon [\mathit{cons}\colon t\alpha, \mathit{nil}\colon \varepsilon]]$ is the least upper bound of the denumerable increasing sequence of approximations depicted in Figure 2.

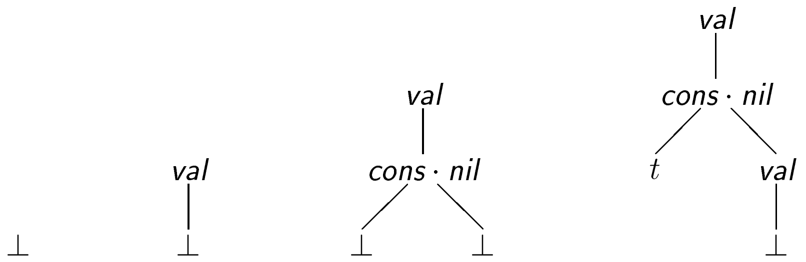

Figure 2: The first approximations of the solution of the equation $\alpha = [\mathit{val}\colon [\mathit{cons}\colon t\alpha, \mathit{nil}\colon \varepsilon]]$.

A *substitution on infinite trees* is a mapping $s : \mathbf{V} \to \mathbf{T}_\infty$. Given a substitution s and an infinite tree ρ, we obtain the infinite tree $s\rho$ as the result of the simultaneous substitution of tree st for each occurrence of type-variable t in ρ. We write $\rho[t := \sigma]$ for the tree $s\rho$, when substitution s assigns the infinite tree σ to type-variable t and leaves unchanged all other type-variables. Any such substitution s is a continuous function from $\mathbf{T}_\infty$ to $\mathbf{T}_\infty$. This fact allows to define a translation function $(\cdot)^*$ from recursive types in $\mathbf{T}_\mu$ into infinite trees in $\mathbf{T}_\infty$ as follows.[1]

$$\begin{array}{rcll} t^* & = & t & \text{for } t \text{ a type-variable,} \\ []^* & = & \varepsilon & \text{for } \varepsilon \text{ the empty sequence of labels,} \\ [l_1\colon \tilde{\alpha}_1, \dots, l_n\colon \tilde{\alpha}_n]^* & = & l_1 \dots l_n(\tilde{\alpha}_1^*, \dots, \tilde{\alpha}_n^*) & \text{for } l_1 \leq \dots \leq l_n \text{ and } n \geq 1, \\ \mu t.\alpha^* & = & \mathbf{fix}(\lambda\rho.\alpha^*[t := \rho]) & \text{for } \rho \text{ an infinite tree.} \end{array}$$

Trees obtained by the above translation are *regular*, that is, they have finitely many subtrees. The results of Braquelaire and Courcelle [2] ensure that every regular tree has a notation in $\mathbf{T}_\mu$. This allows to prove results in $\mathbf{T}_\mu$ using regular trees, and will be particularly useful in establishing the equivalence between TA_μ and the kinded recursive type system, and the existence of principal kinded typings.

Recursive typing assignment. An interpretation of recursive types as infinite trees naturally induces an equivalence relation $\approx$ on $\mathbf{T}_\mu$ by putting $\alpha \approx \beta$ if $\alpha^* = \beta^*$.

Definition 4.2 (Recursive typing assignment system TA_μ) TA_μ is defined by the rules in $\mathrm{TA}_{[]}$ with the addition of the $\approx$-rule, and by replacing the Msg-axiom by the

[1] Naturally, $\tilde{\alpha}^*$ means $\alpha_1^*, \dots, \alpha_n^*$, whenever $\tilde{\alpha} = \alpha_1 \dots \alpha_n$.

MSG_μ-axiom.[2]

$$\approx \quad \frac{P \triangleright \Gamma \cdot x\colon \alpha}{P \triangleright \Gamma \cdot x\colon \beta} \quad (\alpha \approx \beta)$$

$$\text{MSG}_\mu \quad a \lhd l_1 \colon \tilde{v} \triangleright \{a\colon \beta, \tilde{v}\colon \tilde{\alpha}_1\} \quad (\beta \approx [l_1\colon \tilde{\alpha}_1, \ldots, l_n\colon \tilde{\alpha}_n])$$

All syntactical properties of system $\text{TA}_{[]}$ discussed in Section 2 continue to hold when formulated in the extended system TA_μ, notably subject-reduction implying that programs whose names can be assigned recursive types do not run into type mismatch errors during execution. Existence and computability of principal typings is the theme of next section.

5 Principal recursive typings

Not surprisingly, system TA_μ does not possess a notion of principal typings in the sense discussed at the beginning of Section 3. Nonetheless, starting from system TA_μ, and by following the same path that took us from $\text{TA}_{[]}$ to TA_k, we can define a *kinded recursive typing assignment system* $\text{TA}_{\mu k}$.

Kinded recursive typing assignment. Recursion at the level of kinds is achieved by allowing cycles in kindings. In this way, recursive kinds (implicitly) arise as cycles in a kinding, making it unnecessary to have recursive kind expressions.

Definition 5.1 (KINDED RECURSIVE TYPING ASSIGNMENT SYSTEM $\text{TA}_{\mu k}$) $\text{TA}_{\mu k}$ is obtained from system TA_k, with the MSG_k-rule replaced by the $\text{MSG}_{\mu k}$-rule below, together with the $\approx$-rule with formulas $P \triangleright \Gamma$ replaced by $K \vdash P \triangleright \Gamma$, and by allowing cycles in kindings.

$$\text{MSG}_{\mu k} \quad \frac{K \vdash_{\langle\rangle} \beta \triangleright \langle l\colon \tilde{\alpha}\rangle}{K \vdash a \lhd l \colon \tilde{v} \triangleright \{a\colon \gamma, \tilde{v}\colon \tilde{\alpha}\}} \quad (\beta \approx \gamma)$$

As an example, let us derive a typing for the term $\text{Cons}(lfl)$, starting from a (cyclic) kinding $K = \{u\colon \langle \textsf{\textit{cons}}\colon t\,[\textsf{\textit{val}}\colon u]\rangle\}$.

$$\text{OBJ}_k \quad \cfrac{\text{MSG}_{\mu k} \quad \cfrac{K \vdash u \triangleright \langle \textsf{\textit{cons}}\colon t\,[\textsf{\textit{val}}\colon u]\rangle}{K \vdash r \lhd \textsf{\textit{cons}}\colon fl \triangleright \{f\colon t, l\colon [\textsf{\textit{val}}\colon u], r\colon u\}}}{K \vdash l \rhd [\textsf{\textit{val}}\colon (r)\; r \lhd \textsf{\textit{cons}}\colon fl] \triangleright \{f\colon t, l\colon [\textsf{\textit{val}}\colon u]\}}$$

In this case, since name r (the address of the object meant to receive the reply to message *val*) does not appear in an object location position, the recursive nature of the Cons cell manifests itself in a cyclic kinding. Contrast this situation with the term

$$\text{Cons}(lfl),\ \nu r(l \lhd \textsf{\textit{val}}\colon r,\ r \rhd [\textsf{\textit{nil}}\colon \text{NilCase}\ \&\ \textsf{\textit{cons}}\colon (xy)\text{ConsCase}])$$

where we evaluate the Cons cell and branch, according to the result, into terms NilCase or ConsCase. Here, the principal type of l is a recursive type $\alpha = \mu u.[\textsf{\textit{val}}\colon [\textsf{\textit{cons}}\colon tu, \textsf{\textit{nil}}\colon \varepsilon]]$.

[2]This allows to type messages whose target is contained in the message's communication, as in $a \lhd l\colon a$.

The derivation of the subterm $\mathrm{Cons}(lfl)$ is depicted below. Notice that $\alpha \approx [\mathsf{val}\colon [\mathsf{cons}\colon t\alpha, \mathsf{nil}\colon \varepsilon]]$.

$$
\dfrac{\dfrac{\dfrac{[\mathsf{cons}\colon t\alpha, \mathsf{nil}\colon \varepsilon] \;\triangleright\; \langle \mathsf{cons}\colon t\alpha\rangle}{r \lhd \mathsf{cons}\colon fl \;\triangleright\; \{f : t, l\colon \alpha, r : [\mathsf{cons}\colon t\alpha, \mathsf{nil}\colon \varepsilon]\}}\,\textsc{Msg}_{\mu k}}{r \lhd \mathsf{cons}\colon fl \;\triangleright\; \{f : t, l\colon [\mathsf{val}\colon [\mathsf{cons}\colon t\alpha, \mathsf{nil}\colon \varepsilon]], r : [\mathsf{cons}\colon t\alpha, \mathsf{nil}\colon \varepsilon]\}}\,\approx}{\dfrac{l \rhd [\mathsf{val}\colon (r)\; r \lhd \mathsf{cons}\colon fl] \;\triangleright\; \{f : t, l\colon [\mathsf{val}\colon [\mathsf{cons}\colon t\alpha, \mathsf{nil}\colon \varepsilon]]\}}{l \rhd [\mathsf{val}\colon (r)\; r \lhd \mathsf{cons}\colon fl] \;\triangleright\; \{f : t, l\colon \alpha\}}\,\approx}\,\textsc{Obj}_k
$$

Theorem 5.2 *Terms in* $\mathbf{P}$ *are typable in* $TA_{\mu k}$ *if and only they are typable in* TA_{μ}.

Proof: The proof of the corresponding result in $\mathrm{TA}_{[]}$, Theorem 3.4, easily adapts to the recursive setting. The key issue is that the system of regular equations generated by a kinding has a unique solution whose components are regular trees [4], and that each of these trees define a type in $\mathbf{T}_\mu$ [2]. The cases concerning the $\textsc{Msg}_\mu$-axiom and the $\textsc{Msg}_{\mu k}$-rule also need a slight adaptation. □

Typing inference. Existence and computability of principal kinded typings in $\mathrm{TA}_{\mu k}$ (the counterpart of Theorem 3.7 in the recursive setting) is ensured by the existence of a unification algorithm capable of handling recursive types (in the form of regular infinite trees) and cyclic kindings. The proof of the following theorem is out of the scope of this paper and will appear in the author's forthcoming PhD thesis.

Theorem 5.3 *Let* (K, E) *be a kinded set of equations with types in* $\mathbf{T}_\infty$.

i. If (K, E) *is unifiable, it has a most general unifier containing at most the type-variables occurring in* (K, E).

ii. If (K, E) *is regular and unifiable, its most general unifier is regular. It can be effectively computed.*

The very same algorithm discussed at the end of Section 3 to extract the principal TA_k kinded typing of a term can be used to extract the principal $\mathrm{TA}_{\mu k}$ kinded typing, provided that we use a unification algorithm capable of handling recursive types.

Concluding remarks

We presented an extension to the simple type system for the calculus of objects [14]. The new additions encompass a new type constructor, and a new rule in the type system, allowing to type terms with a recurring name structure, in particular, terms denoting numerals, lists and trees. The recursive type system extends the class of typable terms, while preserving basic syntactic properties of the simple system, including subject-reduction, and the existence and computability of principal-typings.

The incorporation of recursive types is one of the two extensions envisioned for the calculus of objects, towards an effective modelling of typed concurrent object-oriented systems. The second extension, comprising a notion of predicative polymorphism in name-passing calculi [12], is intended to capture the idea of a polymorphic class (a stack, for example) from which we can instantiate objects with different types instances of the type of the class (stacks of integers and stacks of lists of boolean values, for example). Also, as noted by a referee, the treatment of recursive types presented in this paper could be easily transposed into the usual λ-calculus with records.

A pragmatic issue is the application of the calculus of objects and its type system to define precise operational semantics and type inference systems for existing object-oriented concurrent programming languages. We picked up ABCL/1 as a case study; the results can be found in [11].

Acknowledgements. I would like to thank Prof. Mario Tokoro for continuous support, Kohei Honda for discussions on types for concurrency, and a referee for invaluable comments on an early draft of the paper.

References

[1] Henk Barendregt and Kees Hemerik. Types in lambda calculi and programming languages. In *3rd European Symposium on Programming*, volume 432 of *LNCS*, pages 1–35. Springer-Verlag, May 1990.

[2] J. P. Braquelaire and B. Courcelle. The solutions of two star-height problems for regular trees. *Theoretical Computer Science*, 30:105–239, 1984.

[3] Felice Cardone and Mario Coppo. Two extensions of Curry's type inference system. In *Logic and Computer Science*, pages 19–75. Academic Press limited, 1990.

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

[5] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal, modular actor formalism for artificial intelligence. In *3rd International Joint Conference on Artificial Intelligence*, pages 235–245, 1973.

[6] Kohei Honda and Mario Tokoro. An object calculus for asynchronous communication. In *5th European Conference on Object-Oriented Programming*, volume 512 of *LNCS*, pages 141–162. Springer-Verlag, 1991.

[7] Kohei Honda and Nobuko Yoshida. On reduction-based process semantics. In *13th Foundations of Software Technology and Theoretical Computer Science*, volume 761 of *LNCS*, pages 371–387. Springer-Verlag, 1993.

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

[9] John C. Mitchell. Type systems for programming languages. In *Handbook of Theoretical Computer Science*, pages 366–458. Elsevier Science Publishers B.V., 1990.

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

[11] Vasco T. Vasconcelos. An operational semantics and a typing system for ABCL/1 based on a calculus of objects. CS 94–001, Keio University, April 1994.

[12] Vasco T. Vasconcelos. Predicative polymorphism in π-calculus. In *5th Parallel Architectures and Languages Europe*, LNCS. Springer-Verlag, July 1994.

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

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

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