93-002

Recursive Types in a Calculus of Objects

Vasco Thudichum Vasconcelos

November 2, 1993

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

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

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

Abstract

We introduce a calculus for concurrent objects composed of labelled methods and communicating via asynchronous message passing, intended to capture some basic features present in most notions of objects. A decidable typing system allows to statically verify that 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 possess an appropriate method.

Then we extend the class of typable terms by introducing infinite types in the form of recursively defined type expressions, along the lines of Cardone and Coppo for the λ-calculus and of Vasconcelos and Honda for the polyadic π-calculus. The new system preserves all the syntactical properties of the simple typing assignment system, including the subject-reduction and the computability of principal typings.

1.

Introduction

Concurrent objects provide a convenient tool to describe concurrent, distributed and open-ended systems. An important issue which arises in such systems is what we might call “security in communications”. Indeed computations are expected to follow smoothly without errors of protocol, and newly created agents are expected to interact with the system in such way that does not endanger the security of the whole system.

Security in communications (not an exclusive of systems of concurrent objects) is often tackled with typing systems that ensure that well typed program will not encounter runtime errors. Furthermore, a type for a program also gives an indication on the correct usage of the program. Nevertheless, most object-oriented concurrent programming languages possess no such notion, leaving to runtime the detection of protocol errors, and often taking drastic measures when an unfortunate, but common, type mismatch error occurs.

Inspired in Milner's polyadic π-calculus [7], Honda's ν-calculus [6] and Hewitt's actor model [5], we developed in [13] a basic calculus for concurrent objects 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. The calculus is built along the trends of name passing calculi, by introducing labelled summations (representing methods) and asynchronous labelled messages selecting a branch in the summation. A typing assignment system assigns types to the free names in (untyped) terms, specifying in some sense the communication protocol of the term, and ensures that objects in well typed programs do not receive messages for which they do not possess an appropriate method.

The present paper extends the class of typable terms by introducing a notion of recursive types, denoting labelled (possibly infinite) trees obtained by the expansion of the recursively defined expression, and by incorporating in the basic typing assignment system a new rule allowing to replace a type by another representing the same tree. All the syntactic properties of the simple type system discussed in [13] extend to the more general recursive setting.

The outline of the paper is as follows. Chapter 2 introduces the calculus for objects. Then, the next two chapters summarize the notions and results in [13]. In particular, chapter 3 presents the simple typing assignment system and chapter 4 studies the notion of principal typings. The following two chapters develop the central theme of the paper. Chapter 5 introduces the notion of recursive types and the recursive typing assignment system, and chapter 6 deals with the notion of principal typings in the recursive setting. The last chapter summarizes the results and points some directions for further work.

2.

The calculus of objects

In this chapter we introduce the calculus of objects, its syntax and semantics. The exposition will be brief since the behavior of programs in not the main topic of the paper.

2.1 Syntax

Terms are built from an infinite set $\mathbf{N}$ of *names* and a set $\mathbf{L}$ of *labels*, by means of six constructors. $\mathbf{N}^*$ is the set of (finite) sequences of names, and $\tilde{v}, \tilde{x}, \ldots$ are arbitrary elements of this set.

Definition 2.1.1 (Syntax) The set of *terms* is given by the following grammar.

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

Names in each $\tilde{x}_i$, $i = 1, \ldots n$ and labels $l_1, \ldots l_n$ are assumed to be pairwise distinct.

Intuitively, 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. 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. If l is the label of some method, say $l_i(\tilde{x}_1).P_i$, in the above object, then, upon reception of the message, the object will behave as P_i with names in $\tilde{x}_i$ replaced by those in $\tilde{v}$.

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 $!P$ (P should be an object) intuitively represents as many copies of P as needed, running in parallel. Finally, $\mathbf{0}$ is a syntactic convention to represent the term which does nothing.

2.2 Semantics

Methods and scope restriction are the binding operators of the calculus. The notion of *free names* in a term is defined from the binding operators, as usual in name passing calculi. Then, the notion of *substitution* of free occurrences of a name x by a name z in a term P, denoted $P[z/x]$, is defined in the usual way. When $\tilde{x}$ is of the same length as $\tilde{z}$ and names in $\tilde{x}$ are pairwise distinct, $P\{\tilde{z}/\tilde{x}\}$ denotes the *simultaneous substitution* of 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 [13] 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, the substitution of the message contents by the method's formal parameters, and the execution of the method body.

Definition 2.2.1 (Message application) Let $C = l(\tilde{v})$ 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. Message application is defined by

$$M \bullet C \rightarrow P_k\{\tilde{v}/\tilde{x}_k\}$$

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

In order to simplify the statement of the reduction relation, we notice that any term P has an equivalent term P' of the form $\nu\tilde{u}(P_1, \dots P_n)$, where terms P_i, $i = 1, \dots n$, called *base terms*, are either messages or objects (eventually replicated).

Definition 2.2.2 (Reduction) *One-step reduction*, denoted by $\rightarrow$, is the smallest relation generated by the following rules.

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

$$\text{Comm} \qquad \frac{M \bullet C \rightarrow P}{\nu\tilde{u}(\partial, a \lhd C, a \rhd M, \partial') \rightarrow \nu\tilde{u}(\partial, P, \partial')}$$

In rule Comm, ∂ and ∂' represent concurrent composition of base-terms, M is a collection of methods of the form $[l_1(\tilde{x}_1).P_1 \ \& \cdots \& \ l_n(\tilde{x}_n).P_n]$ and C is a communication of the form $l(\tilde{v})$.

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

3.

Simple typing assignment

In this chapter 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.

3.1 Types for names

Types are built from an infinite set $\mathbf{V}$ of *type variables* and the set $\mathbf{L}$ of labels introduced in Chapter 2, by means of a single constructor.

DEFINITION 3.1.1 (SIMPLE TYPES) The set $\mathbf{T}$ of *types* is defined by grammar

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

where $l_1, \ldots l_n$ represent 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 $\mathbf{T}^*$.

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 types $\tilde{\alpha}_i$.

3.2 Simple typing assignment

Type assignments are formulas $x : \alpha$, 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 $\{x_1 : \alpha_1, \ldots x_n : \alpha_n\}$, where no two assignments have the same name as subject. If $\tilde{x} = x_1 \cdots x_n$ is a sequence of distinct names and $\tilde{\alpha} = \alpha_1 \cdots \alpha_n$ a sequence of types, we will often write $\tilde{x} : \tilde{\alpha}$ for $\{x_1 : \alpha_1, \ldots x_n : \alpha_n\}$, and $\Gamma \cdot \tilde{x} : \tilde{\alpha}$ for $\Gamma \cup \tilde{x} : \tilde{\alpha}$, provided names in $\tilde{x}$ do not occur in typing Γ.

DEFINITION 3.2.1 (TYPING COMPATIBILITY) Typings Γ and Δ are *compatible*, denoted $\Gamma \asymp \Delta$, if $\alpha = \beta$ for all $x : \alpha \in \Gamma$ and $x : \beta \in \Delta$

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

DEFINITION 3.2.2 (TYPING ASSIGNMENT SYSTEM TA) TA is defined by the following rules.

$$\textsc{Nil} \qquad \vdash \mathbf{0} \triangleright \emptyset$$

$$\textsc{Msg} \qquad \vdash a \lhd l(\tilde{v}) \triangleright \{\tilde{v} : \tilde{\alpha}, a : [l : \tilde{\alpha}, \ldots]\}$$

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

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

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

We conclude the chapter with a brief overview of some important properties of the typing assignment system. 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. Typings for typable terms assign types to all free names in the term and, whenever a term is typable, there is a derivation assigning types to only the free names in the term, thus yielding the "smallest" possible typing for the term.

Subject-reduction ensures that a typing for a term does not change as the term is reduced. As a corollary, we have that typable terms do not run into runtime errors. We say a term P contains a possible *runtime error* if it may be reduced to a term Q of the form $\nu\tilde{u}(\ldots a \rhd M, a \lhd C, \ldots)$ 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 chapter.

4.

Principal typings

There are two properties we would like our typing assignment system to verify (cf. [2]).

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

The solution to both problems is usually obtained via the notions of principal typings and subsumption (cf. [8]). We say a typing Γ *subsumes* Γ' if every term with typing Γ' also has a typing Γ. A *principal typing* for a term P is a typing for P that subsumes all others. If a typing system has a computable notion of principal typings and subsumption is decidable, then the decidability of typing inference $\vdash P \triangleright \Gamma?$, reduces to computing the principal typing for P and verifying if it subsumes Γ. Computability of typing existence $\vdash P \triangleright ?$, amounts to compute the principal typing for P, since this is a well-typing for P.

In λ-calculi subsumption is often connected with substitution of type variables for types. When labels are present to model certain characteristics of objects, subsumption also takes into account the number and nature of labels in a type. Unfortunately, system TA does not possess such a notion of subsumption. The reason seems to lie in the fact that names present in terms but never appearing in object location positions, can never be assigned "closed" record types, for we can never know the types of *all* the messages the object is able to receive (see [13] for a concrete example). Nevertheless, principal typings and subsumption may be recovered with a notion of constraints on the types a variable may be substituted for, by using Ohori's kinds [9].

4.1 Kinds and kind assignment

Intuitively, kinds describe constraints on the substitution of type variables; a kind of the form $\langle l_1 : \tilde{\alpha}_1, \ldots l_n : \tilde{\alpha}_n \rangle$ denotes the subset of record types containing *at least* the components $l_1 : \tilde{\alpha}_1, \ldots l_n : \tilde{\alpha}_n$.

Definition 4.1.1 (Kinds) The set $\mathbf{K}$ of *kinds* 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 $n \geq 0$. We use $k, k', \dots$ to range over $\mathbf{K}$.

Kind assignments are formulas $t : k$, for t a type variable and k a kind. *Kindings*, denoted by $K, K' \dots$, are sets of kind assignments where no two assignments have the same type variable as subject. Furthermore, we require kindings to be *acyclic* and *closed*. 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 $n \geq 1$. A set of kind assignments is said to be closed if each type variable occurring in a predicate also occurs as the subject of some kind assignment.

The inexistence of cycles in typings is what keeps types finite, when we replace type variables by types constrained to some kinding. The requirement that kindings are closed ensures that all type variables are kinded and will be particularly important in the recursive setting, making a kinding a regular system of equations. In particular, unconstrained (completely free) type variables have the empty kind $\langle\rangle$.

Definition 4.1.2 (Kind assignment to types) A type α *has a kind k under a kinding K*, if the statement $K \vdash \alpha : k$ can be derived from the following rules.

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

4.2 Kinded typing assignment

Kinded typing assignments are expressions of the form $K \vdash P \triangleright \Gamma$, where all type variables in typing Γ occur in kinding K. We will write $K \vdash_k P \triangleright \Gamma$ if the statement $P \triangleright \Gamma$ is provable from kinding K, using the rules and axioms of TA_k below.

Definition 4.2.1 (Kinded typing assignment system TA_k) TA_k is defined by the rules in TA with sequents of the form $\vdash P \triangleright \Gamma$ replaced by $K \vdash P \triangleright \Gamma$, and by replacing rule Msg for rule

$$\textsc{Msg}_k \quad \frac{K \vdash \beta : \langle l : \tilde{\alpha} \rangle}{K \vdash a \lhd l(\tilde{v}) \triangleright \{\tilde{v} : \tilde{\alpha}, a : \beta\}}$$

We can easily prove that TA_k is correct with respect to TA.

Theorem 4.2.2 *A term P is typable in TA_k if and only if it is typable in TA.*

PROOF: To prove the forward implication, we note that deductions in TA_k can be mapped into deductions in TA, by replacing kinded type variables by types constrained to the kinding. The fact that kindings are acyclic, ensures that the replacement always terminates. The reverse implication follows from the fact that 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 (see [13] for details). □

4.3 Subsumption and principal kinded typings

A *substitution on types* is a mapping $s : \mathbf{V} \rightarrow \mathbf{T}$ 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. Following Ohori [9], we say a kinded substitution (K', s) *respects* a kinding K if and only if $K' \vdash st : sk$, for all $t : k \in K$.

A kinded typing is a pair (K, Γ) composed of a kinding K and a typing Γ. We say that a kinded typing (K, Γ) *subsumes* (K', Δ) if there is a substitution s such that (K', s) respects K and $s\Gamma \subseteq \Delta$. A *principal kinded typing* for a term P is a kinded typing for P which subsumes all other kinded typings for P. The good news about TA_k is that principal typings are computable.

THEOREM 4.3.1 (EXISTENCE AND COMPUTABILITY OF PRINCIPAL TYPINGS)
If P is typable then there exists a principal kinded typing for P. It can be effectively computed.

4.4 Typing reconstruction

The computability of principal typings is assured by the existence of a kinded unification algorithm which computes the most general kinded typing of a kinded set of type equations.

Again following Ohori [9], we say 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 4.4.1 (KINDED UNIFICATION [9]) *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.*

An efficient algorithm to extract the principal kinded typing of a term follows the derivation 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 set of equations is then submitted to the kinded unification algorithm. The algorithm follows the lines of that of Wand for the simply typed λ-calculus [14] and that of Vasconcelos and Honda for the polyadic π-calculus [12], and can be found in [13].

5.

Recursive types and recursive typing assignment

There are many meaningful and useful terms that cannot be typed in the simple typing assignment system describe so far. Terms representing natural numbers and lists are two examples of a more general class of terms representing recursive data structures. Lists, for example, can be built from two basic constructors, Nil and Cons, capable of receiving one single eval message. To such a message, Nil objects reply nil, stating "I am a Nil object". On the other hand, Cons objects reply $\mathsf{cons}(f\,s)$, stating "I am a Cons object, and here is my first and second elements".

$$\begin{array}{rcl} \mathrm{Nil}(l) & \stackrel{\text{def}}{=} & l \triangleright [\mathsf{eval}(r).r \triangleleft \mathsf{nil}] \\ \mathrm{Cons}(l f s) & \stackrel{\text{def}}{=} & l \triangleright [\mathsf{eval}(r).r \triangleleft \mathsf{cons}(f\,s)] \end{array}$$

Actual lists are built from these two constructors, by appropriately hiding the second element of the Cons objects. So, for example a list of two elements a and b, located at name l, can be written as

$$\nu s(\mathrm{Cons}(l\,a\,s), \nu s'(\mathrm{Cons}(s\,b\,s'), Nil(s')))$$

In conventional usages of lists, the type of the list itself and that of its second element coincide. However, it is easy to see that no typing can assign to names l and s in an object $\mathrm{Cons}(lfs)$ the same (finite) type, for we would need a type α such that $\alpha = [\mathsf{eval}\!:\![\mathsf{cons}\!:\!\beta\alpha, \ldots]]$, for some type β.

5.1 Recursive types

Following [3, 12], we can use equations between types, whose solutions will be infinite types, that is expressions whose construction trees are infinite labelled trees. So, for example, to an equation of the form

$$\alpha = [\mathsf{eval}\!:\![\mathsf{cons}\!:\!t\alpha, \mathsf{nil}\!:\!\varepsilon]]$$

where ε denotes the empty sequence of types, corresponds the tree depicted in figure 5.1.

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 a type. Intuitively, a type $\mu t.\alpha$ may be

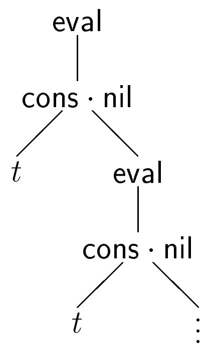

Figure 5.1 The infinite tree associated with equation $\alpha = [\mathsf{eval}:[\mathsf{cons}:t\alpha, \mathsf{nil}:\varepsilon]]$.

considered as a finite notation for the possibly infinite tree that solves the equation $t = \alpha$.

Definition 5.1.1 (Recursive types) The set of *recursive types* $\mathbf{T}_\mu$ is defined by adding to the syntax of simple types in definition 3.1.1, a production $\mu t.\alpha$, for any type variable t and any type α in $\mathbf{T}_\mu$.

5.2 Recursive types as infinite trees

We assume the set $\mathbf{L}$ of labels is equipped with a ranking function $rank : \mathbf{L} \rightarrow \mathcal{N}$, where $\mathcal{N}$ is the set of natural numbers. Then we build the set of *tree labels* — a ranked alphabet $\mathbf{F}$ — as the union of the set $\mathbf{V}$ of type variables and the set of strings of pairwise distinct labels in $\mathbf{L}$, and define the rank of a type variable as 0 and the rank of a sequence of labels $l_1 \cdots l_n$, $n \geq 0$, as $\sum_{i=1}^{n} rank(l_i)$.

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

Strings of positive integers describe paths in a tree; the nodes of the tree being either type variables of strings of pairwise distinct labels, representing records. If $\rho_1, \ldots \rho_n$ are n trees, and the rank of tree label f is n, then we write $f[\rho_1 \cdots \rho_n]$ for the tree ρ such that

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

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

i. $\perp \sqsubseteq \rho$, for all infinite trees ρ; and

ii. $f[\rho_1 \cdots \rho_n] \sqsubseteq f[\rho_1' \cdots \rho_n']$, if $\rho_1 \sqsubseteq \rho_1', \ldots \rho_n \sqsubseteq \rho_n'$, for all $n \geq 0$.

It is easy to see that the set $\mathbf{T}_\infty$ of infinite trees 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 least solution of the equation $\alpha = [\mathsf{eval}:[\mathsf{cons}:t\alpha, \mathsf{nil}:\varepsilon]]$ is the least upper bound of the denumerable increasing sequence of approximations depicted in figure 5.2.

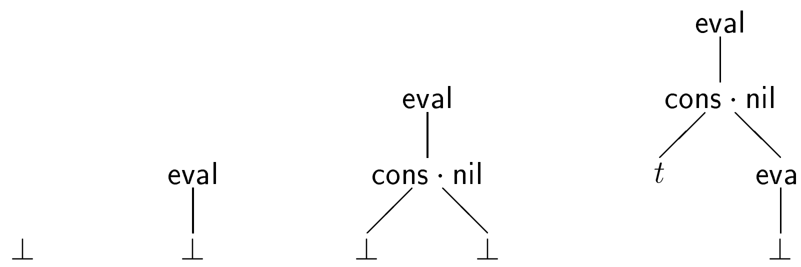

Figure 5.2 The first approximations of the solution of equation $\alpha = [\mathsf{eval}:[\mathsf{cons}:t\alpha, \mathsf{nil}:\varepsilon]]$.

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

$$
\begin{array}{rcll}
t^* & = & t & \text{for } t \text{ a type variable} \\
[l_1:\tilde{\alpha}_1, \ldots l_n:\tilde{\alpha}_n]^* & = & l_1 \cdots l_n[\tilde{\alpha}_1^* \ldots \tilde{\alpha}_n^*] & \text{for } n \geq 0 \\
\mu t.\alpha^* & = & \mathbf{fix}(\lambda\rho.\alpha^*[t := \rho]) & \text{for } \rho \text{ an infinite tree}
\end{array}
$$

An important fact about trees associated with recursive types is that they are *regular*, that is, they have infinitely many subtrees. Another property of the translation is that we can always decide which tree ρ_i "belongs" to each label l_j in a tree $l_1 \cdots l_n[\rho_1, \ldots, \rho_m]$, given that $\mathbf{L}$ is ranked. This two properties ensure the existence of a reverse translation from regular trees in $\mathbf{T}_\infty$ into recursive types in $\mathbf{T}_\infty$.

The reader may have noticed that the translation of recursive types into trees is not unique, since different orderings on the components in a record yield different trees. If uniqueness of translation is an issue, we can set up a total ordering on $\mathbf{L}$ (say the lexicographic order) and require that record types presented to the translation are ordered.

5.3 Recursive typing assignment

An interpretation of recursive types as infinite trees naturally induces an equivalence relation $\approx$ on $\mathbf{T}_\mu$ by putting

$$\alpha \approx \beta \quad \text{if and only if} \quad \alpha^* = \beta^*$$

Definition 5.3.1 (Recursive typing assignment system TA_μ) TA_μ is defined by the rules in definition 3.2.2 with the addition of rule $\approx$, and by replacing rule Msg by rule Msg_μ.[1]

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

$$\mathrm{Msg}_\mu \qquad \vdash a \lhd l(\tilde{v}) \;\triangleright\; \{\tilde{v} : \tilde{\alpha}, a : \beta\} \quad (\beta \approx [l : \tilde{\alpha}, \ldots])$$

All the syntactical properties of system TA discussed in chapter 3 continue to hold when formulated in the extended system TA_μ, notably the Subject-reduction property which implies that programs whose names can be assigned recursive types do not run into type mismatch errors during execution. The property of existence of principal typings is left to the next chapter.

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

6.

Principal recursive typings

Not surprisingly, system TA_μ does not possess a notion of principal typings, in the sense discussed in the beginning of Chapter 4. Nonetheless, we can define a *kinded recursive typing assignment system*, starting from system TA_μ by following the same method that took us from TA to TA_k.

6.1 Kinded recursive typing assignment

To allow recursion at the level of kindings we simply allow cycles in kindings. In this way, recursive kinds (implicitly) arise as cycles in a kinding, making it unnecessary to have explicit recursion at the level of kindings. Since we now have one more constructor to build types, we extend the definition of kind assignment to types, with the following rule.

$$K \vdash \mu t.\alpha : k \qquad \text{if} \qquad K \vdash \alpha[\mu t.\alpha/t] : k$$

Definition 6.1.1 (Kinded recursive typing assignment system $\mathrm{TA}_{\mu k}$)
$\mathrm{TA}_{\mu k}$ is obtained from system TA_k by adding rule ($\approx$) with sequents of the form $\vdash P \triangleright \Gamma$ replaced by $K \vdash P \triangleright \Gamma$, and by allowing cycles (but not recursive types) in kindings.

As an example, let us derive a typing for the term $\mathrm{Cons}(lfl)$, starting from a (cyclic) kinding $K = \{t\!:\!\langle\rangle, u\!:\!\langle \mathsf{cons}\!:\!t\,[\mathsf{eval}\!:\!u]\rangle\}$.

$$\textsc{Obj}_k\ \dfrac{\textsc{Msg}_k\ \dfrac{K \vdash u : \langle \mathsf{cons}\!:\!t\,[\mathsf{eval}\!:\!u]\rangle}{K \vdash_{\mu k} r \lhd \mathsf{cons}(fl) \triangleright \{f\!:\!t, l\!:\![\mathsf{eval}\!:\!u], r:u\}}}{K \vdash_{\mu k} l \rhd [\mathsf{eval}(r).r \lhd \mathsf{cons}(fl)] \triangleright \{f\!:\!t, l\!:\![\mathsf{eval}\!:\!u]\}}$$

The kinded typing $(K, \{f:t, l\!:\![\mathsf{eval}:u]\})$ is indeed principal for $\mathrm{Cons}(lfl)$. It is interesting to note that, in the absence of a "user" of the list, the recursive nature of the list manifests itself in a cyclic kinding. In contrast, a term of the form

$$\nu r(\mathrm{Cons}(lfl), l \lhd \mathsf{eval}(r), r \rhd [\mathsf{nil}.\mathbf{0}\ \&\ \mathsf{cons}(xy).\mathbf{0}])$$

has a principal typing assigning to l the recursive type $\mu u.[\mathsf{eval}:[\mathsf{cons}:tu, \mathsf{nil}:\varepsilon]]$.

We still have to show that system $\mathrm{TA}_{\mu k}$ is correct with respect to system TA_{μ}. The proof that terms typable in TA are also typable in TA_k— the forward implication of theorem 4.2.2 — easily extends to the recursive setting. However the proof that terms typable in $\mathrm{TA}_{\mu k}$ are also typable in TA_{μ} is complicated by the fact that kindings in $\mathrm{TA}_{\mu k}$ may be cyclic.

THEOREM 6.1.2 *A term P is typable in $TA_{\mu k}$ if and only if it is typable in TA_{μ}.*

PROOF: The reverse implication follows by a simple extension of the corresponding proof in [13]. To prove the forward implication, assume $K \vdash_{\mu k} P \triangleright \Gamma$, for some kinded typing (K, Γ). We build a typing Γ' from (K, Γ) and show that $\vdash_{\mu} P \triangleright \Gamma'$.

We start by building a system S of regular equations containing an equation of the form $t = l_1 \cdots l_n[\tilde{\alpha}_1^* \cdots \tilde{\alpha}_n^*]$ for each kind assignment $t : \langle l_1 : \tilde{\alpha}_1, \cdots l_n : \tilde{\alpha}_n \rangle$ in K. Since types occurring in a kind are not recursive, each $\tilde{\alpha}_i$, for $i = 1, \ldots n$, is a sequence of finite trees. Also, by the definition of kindings, all unknowns t occurring in the right hand side of the equations also occur at the left hand side of some equation. Finally, given the form of kinds, no right hand side is a unknown. Hence S is a *generalized regular Greibach system*, and as such has a unique solution in T_{∞}^{n} whose components are regular trees [4].

To obtain Γ', we translate the infinite trees in the solution of S back into recursive types (this can be easily achieved since the trees are regular) and replace in Γ occurrences of type variables by their corresponding recursive types. We conclude the proof showing, by induction on the length of derivations, that $\vdash_{\mu} P \triangleright \Gamma'$. Deductions terminating with rules other than MSG_k follow directly from the induction hypothesis and the corresponding inference rules. We concentrate on derivations terminating with rule MSG_k. So suppose that $K \vdash \beta : \langle l : \tilde{\alpha} \rangle$. We have three cases depending on the nature of β.

i. β is a type variable. Depending on whether β occurs in $\tilde{\alpha}$ or not, the type associated with β in the solution of S can be of two forms.
   a) $[l : \tilde{\alpha}']$, where $\tilde{\alpha}'$ is the sequence of the types associated with $\tilde{\alpha}$ in the solution of S. We then have $\tilde{v} : \tilde{\alpha}'$ and $a : \beta \approx [l : \tilde{\alpha}']$. The result follows by rule MSG_{μ}.
   b) $\mu t.[l : \tilde{\alpha}']$, in which case the sequence of types associated with $\tilde{\alpha}$ is $\tilde{\alpha}'' = \tilde{\alpha}'[\mu t.[l : \tilde{\alpha}']/t]$. We then have $\tilde{v} : \tilde{\alpha}''$ and $a : t \approx [l : \tilde{\alpha}'']$. The result follows by rule MSG_{μ}.

ii. β is a record type. The result follows directly from rule MSG_{μ}.

iii. β is a recursive type $\mu t.\alpha$. We have $K \vdash \alpha[\mu t.\alpha/t] : \langle l : \tilde{\alpha} \rangle$ by the kind assignment rules, and we apply case ii or case iii, depending on the nature of type $\alpha[\mu t.\alpha/t]$.

□

6.2 Typings reconstruction

The decidability of type inference and computability of typings in $\mathrm{TA}_{\mu k}$ is ensured by the fact that Ohori's kinded unification algorithm can be extended to handle recursive types (as infinite trees implemented with direct acyclic graphs) by using the UNION-FIND algorithm [1]. Such an algorithm has been in use for over six months; we conjecture that it is correct, but the details remain to be checked. As such, the very same algorithm discussed at the end of Chapter 4 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 we use a unification algorithm capable of handling recursive types and cyclic kindings.

7.

Concluding remarks

We extended the class of typable terms by introducing a new type constructor and a new rule in the simple typing assignment system presented in [13]. The new constructor allows to type objects with a recurring name structure, in particular objects denoting recursive data-structures such as lists and trees. The new additions preserve the syntactic properties of the typing assignment system, among which we cite the subject-reduction property and the existence 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, incorporating the notion of predicative polymorphism, is intended to capture the idea of an abstract class (say a stack) from which we can instantiate different objects (say a stack of integers and a stack of lists of boolean values), along the lines of [11].

A pragmatic issue is the application of the calculus and its typing system to define precise operational semantics and typing inference systems for existing concurrent object-oriented programming languages. We picked up ABCL/1 as a case study; the results can be found in [10].

Acknowledgements The author would like to thank Prof. Mario Tokoro for continuous support and Kohei Honda for cooperation in the quest for types for concurrency.

Bibliography

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

[2] Henk Barendregt and Kees Hemerik. Types in lambda calculi and programming languages. In *Proceedings of ESOP'90*, Springer-Verlag, 1990. LNCS.

[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. Control structures as patterns of passing messages. *Artificial Inteligence*, 8(3):323–364, 1977.

[6] 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.

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

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

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

[10] Vasco T. Vasconcelos. Operational semantics and a type system for ABCL/1 based on a calculus for objects. November 1993. Keio University.

[11] Vasco T. Vasconcelos. Predicative Polymorphism in π-Calculus. November 1993. Keio University. Submitted.

[12] 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. Also available as Keio University Report CS-92-004.

[13] Vasco T. Vasconcelos and Mario Tokoro. A typing system for a calculus of objects. In S. Nishio and A. Yonezawa, editors, *Int. Symp. on Object Technologies for Advanced Software, ISOTAS'93*, pages 460–474, Springer-Verlag, November 1993. LNCS 742.

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