A Process-Calculus Approach to Typed Concurrent Objects

Vasco Thudichum Vasconcelos

Keio University

Department of Computer Science

3–14–1 Hiyoshi Kohoku-ku Yokohama 223

Japan

November 1994

Submitted as a partial fulfillment for the

Degree of Doctor of Philosophy in Computer Science

ABSTRACT. I propose an untyped name-passing calculus and a series of decidable type systems assigning types to names in terms. The calculus, designed according to the modern trends in name-passing calculi, is intended to capture the basic concepts of concurrent objects. The type systems, developed along the lines of those for λ-calculi, provide for partial-specification in the form of communication protocols for programs, and for partial-correctness wherein typable terms are assured not to run into errors at execution time.

Terms are built from names and labels by means of five constructors, the innovations being guarded labelled-sums describing objects composed of labelled methods and asynchronous labelled messages selecting a method in the target object. The remaining constructors are fairly standard in name-passing calculi and comprise concurrent composition, local name creation (or scope restriction), and replication.

The type systems assign types to names, and typings (that is, sets of name-type pairs) to terms. Simple types are built from type-variables by means of a single record constructor that uses the labels in the syntax of the calculus. To realize typing terms containing circular name-structures, a recursive type constructor is introduced, recursive types being viewed as regular (that is, having finitely many subtrees) infinite trees. The main properties of these type systems are subject-reduction (implying that typable terms do not encounter errors at execution time), decidability of typing assignment, and computability of typing inference. Nonetheless, no notion of typing subsumption, and hence of principal typings, seems to be possible in these systems. I use constraints on the types that may replace a given type-variable, and study the problem of unifying infinite trees with variables subject to constraints.

From a pragmatic point of view, the calculus provides the building blocks necessary to describe the formal operational semantics of concurrent object-oriented programming languages; more complex notions of objects are easily encoded into the basic calculus. Simultaneously, the type systems developed can be used to derive type systems for high-level programming languages. From a theoretical point of view, this work brings into the name-passing paradigm (not restricted to the calculus presented, but encompassing the π-calculus and the ν-calculus) the notions of types and type assignment, thus opening the possibility to explore, in the setting of name-passing calculi, the wide body of results on type systems for λ-calculi.

Acknowlegements

This thesis is the culmination of a long educational journey depending on a non-enumerable series of contributions. Let me try to pinpoint a few.

Professor Mario Tokoro, my supervisor, made it all happen. Since I met him in Lisbon in the summer of 1988, and in particular after coming to Japan in the spring of the following year, he has been an endless source of support and friendship, ranging through aspects of life as a foreigner in Japan, to guidance in the academic world. He introduced me the world of objects, and his comments and encouragement at all stages of the work in which this thesis culminates, were invaluable. I must also thank Professor Norihisa Doi for constructive comments on several occasions, Professors Masakazu Nakanishi and Morio Nagata for comments on this thesis, and Professor Yuichiro Anzai for long-term encouragement.

Special thanks to Kohei Honda for the work we developed together, but above all, for showing what enthusiasm and dedication to research can bring about. He was a true mentor during the past five years. Many thanks to all the people at the Tokoro Lab: to Nobuko Yoshida, Kaku Takeuchi, and Makoto Kubo for the Wednesday seminars on concurrency, for fruitful discussions and interactions; to Ichiro Satoh and to Fumihiko Chimura for being always there, for helping with the Japanese language, and for friendship; to Tatsuo Minohara for showing how to be unique in a group; and to the remaining people in the laboratory for friendship and help with the everyday life.

My thanks to Professor Carl Hewitt for introducing me the universe of actors. Concurrency come into my life with his visit to Keio University in 1989. I hope we will perceive a reflex of his actors in this thesis. Thanks also to Professor Atsushi Ohori for suggesting the idea of using the notion of kinds and the fruitful discussions and interactions that followed. I should also thank Professor Akinori

Yonezawa for support, insightful discussions, and for believing in my work. Special thanks to Professor Peter Wegner, Professor Cliff Jones, and Professor Hikoe Enomoto for comments on my work.

Going back in time, I must thank the excellent environment at the Computer Science Department of Universidade Nova de Lisboa where I graduated. In particular, my deepest thanks to Professor Antonio Porto and Professor Luis Monteiro for long-term support and friendship.

My hearty thanks to my family, my mother in particular, for love and support all the way along this long journey, but above all for the motivating environment I always had at home. Finally, my deepest thanks to my wife who endured five years of Japan, late-evenings arrivals and lost weekends, for companionship, love, and support. João, obrigado.

Yokohama, November the 4th 1994

This thesis includes a revised version of Chapter 6.

Contents

CHAPTER 1

Introduction

1. Aim

This study aims at providing solid operational semantics and type systems for concurrent object-oriented programming languages.

The world of objects is wide, the nomenclature is varied, and where the frontiers of objects lie is not agreed upon. There are however a few features I consider essential in concurrent object-oriented computing; they are

1. primitive concurrency,
2. communication via asynchronous message passing,
3. labelled method invocation,
4. dynamic creation of local names.

Primitive concurrency seems indispensable in trying to model concurrent languages. Sequentiality need not (and should not) be primitive; it can arise as causal dependency among concurrent actions. *Communication via asynchronous message passing* means that there is no synchronization between the sender and the receiver of a message. Synchronous message passing (or remote procedure call), where the sender waits for a reply, can be described with a pair of asynchronous messages. *Labelled method invocation* is a distinctive characteristic of objects, as opposed to, for example, positional invocation or left/right injections. Finally, *dynamic creation of names* (or of object identifiers) provides for much of the flexibility of concurrent languages, allowing changes in the topology of interaction, and to safeguard local computations from external interferences.

There are some notions usually attributed to objects that I deliberately decided not to consider in this basic study. The most prominent are eventually classes of objects, sub-classes and inheritance. I believe these notions are not fundamental in concurrent objects, and that they can be incorporated at a later stage built upon of some primitive notion of concurrent objects. Also, input-output behaviour (function-application in particular) and client-server architectures can all be described by the primitives presented in this thesis.

The second objective of this thesis is to study type systems for name-passing calculi that provide for *partial correctness* and *partial specification.* Partial specification in the sense that types for programs usually give some indication about the usage of the program; partial correctness in that typable programs do not encounter errors at execution time.

The reception of messages for which objects have no appropriate method is among the most common problems in the execution of object-oriented programs. Two kinds of problems may occur: the target object may not possess the particular method the message selects, or it may have the method but the number and type of arguments in the message do not match those provided by the method.

The free names in a program constitute the interaction ports of the program; types for these names specify in some sense the interface of the program. It is in this sense that the free names and their types constitute a form of partial specification for the program.

2. Overview

This thesis comprises the proposal of an untyped name-passing calculus (chapter 2), the study of type systems for the calculus (chapters 3 to 5), and an investigation on unification of kinded infinite trees (chapter 6). Chapter 7 discusses related works (the π- and the ν-calculus is particular), extensions to the basic system, and applications to concrete programming languages.

The calculus of objects. The calculus of objects is a name-passing calculus built along the lines of Milner, Parrow, and Walker's π-calculus [**40**], its polyadic form [**39**], and Honda's ν-calculus [**28**]. Name-passing calculi arise from process algebras, CCS [**36**] in particular, by allowing names (or channels) to carry names.

Hewitt's actor model of computation [**2, 24, 25**] was another source of inspiration for the calculus of objects. The three things an actor can do upon reception of a message—send messages, create new actors, and become a new actor—have their counterparts in the calculus of objects. The idea of selection of methods through labels is also present in the actor model.

The distinctive characteristics of the calculus are *labelled sums* denoting methods in objects, and *labelled asynchronous messages* selecting a particular method in the target object. If a and v are two names, and l_i is a label, then the term

$$a \lhd l_i : v$$

denotes a *message* targeted at (some object located at) a, and carrying a communication labelled by l_i with contents v. On the other hand, if $a, x_1, \dots, x_n$ are names, $l_1, \dots, l_n$ are pairwise distinct labels, and $P_1, \dots, P_n$ are arbitrary processes, then the term

$$a \rhd [l_1 : (x_1)\, P_1 \,\&\, \cdots \,\&\, l_n : (x_n)\, P_n]$$

denotes an *object* located at name a, and composed of methods $l_i : (x_i)\, P_i$, where x_i is the formal argument of the method. The result of the interaction of the above message and object is the process P_i with name x_i replaced by name v, provided that $1 \leq i \leq n$. Interaction between a message and an object constitutes the basic computing mechanism of the calculus, lying at the basis of the *reduction* relation.

This constitutes the core of the monadic calculus; the polyadic version that I shall be dealing with is obtained from the monadic one by allowing messages to carry arbitrarily long sequences of names instead of a single name, and methods to have sequences of names as parameters. The remaining constructors are fairly standard in process calculi. They are: *concurrent composition* P, Q running processes P and Q in parallel, *scope restriction* $\nu x P$ allowing creating a new name x and its use in process P, and *replication* $!P$ denoting an unbounded number of copies of process P, providing in particular for infinite behaviour.

The type systems. The calculus is untyped, or more precisely, implicitly typed. In contrast with λ-calculi where terms are assigned types, in name-passing calculi it is names which are assigned types; terms are assigned sets of name-type pairs called *typings* [**65**].

Simple types are built from *type-variables* and the labels in terms, by means of a single constructor: a *record*. A type-variable is a type, and, if $l_1, \dots, l_n$ are n

pairwise distinct labels, and $\tilde{\alpha}_1, \ldots, \alpha_n$ are n sequences of types, then

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

is also a type. Intuitively, if names $x_1, \ldots, x_n$ in process $a \triangleright [l_1 : (x_1)\, P_1 \,\&\, \cdots \,\&\, l_n : (x_n)\, P_n]$ have types $\alpha_1, \ldots, \alpha_n$, then name a has a type $[l_1 \colon \alpha_1, \ldots, l_n \colon \alpha_n]$. If P is a process and Γ is a typing, I write $P \blacktriangleright \Gamma$ to denote the assignment of Γ to P.

The *type systems* (more precisely, the *typing assignment systems*) studied in this thesis are formal systems in the sense of Hindley and Seldin [**26**], and similar to type systems for λ-calculi.

The first system I study, the *simple type system* $\mathrm{TA}_{[]}$, comprises one rule per constructor in the syntax of the calculus, plus a Weak-rule allowing addition of a new name-type pair to a typing. I write $\vdash_{[]} P \blacktriangleright \Gamma$ to mean that the assignment of typing Γ to process P is deducible in system $\mathrm{TA}_{[]}$. The main properties of system $\mathrm{TA}_{[]}$ (and of those that follow) include the ones below:

1. Subject-reduction (SR)
   If $\vdash_{[]} P \blacktriangleright \Gamma$ and P reduces to Q, then $\vdash_{[]} Q \blacktriangleright \Gamma$.
2. Decidability of typing assignment ($\vdash P \blacktriangleright \Gamma?$)
   Given a term P and a typing Γ, it can be decided whether $\vdash_{[]} P \blacktriangleright \Gamma$.
3. Computability of typing inference ($\vdash P \blacktriangleright ?$)
   Given a term P it can be decided whether there is a typing Γ such that $\vdash_{[]} P \blacktriangleright \Gamma$, and moreover such Γ can be computed from P.

The second type system studied in this thesis arises from the observation that names containing *recurring name-structures* cannot be typed in $\mathrm{TA}_{[]}$. Recurring name-structures arise in terms where a name carries a name that carries a name that ... eventually carries the initial name. The simplest example of a recurring name-structure shows up in the message $a \triangleleft l \colon a$; more interesting examples involve the encodings of inductively defined data-structures such as natural numbers, lists, and trees.

The syntax of types is then extended with a *recursive type-constructor*, so that, if t is a type-variable, and α a type, then also $\mu t.\alpha$ is a type. Recursive types are interpreted as *regular infinite trees*, a type $\mu t.\alpha$ being defined as the only tree that solves the equation $t = \alpha$. The *recursive type system* $\mathrm{TA}_{[]\mu}$ is obtained from system $\mathrm{TA}_{[]}$ by adding one rule allowing replacement of a type in a typing by another representing the same infinite tree; it preserves all the properties of the simple system studied in this thesis.

Properties $\vdash P \blacktriangleright \Gamma?$ and $\vdash P \blacktriangleright ?$ are usually built on the notion of *typing subsumption*, and the derived notion of *principal typing*. System $\mathrm{TA}_{[]}$, however, possesses no notion of typing subsumption based on substitution of types for type-variables and on addition of components to records. This is because, for example, if the only occurrence of a name a in some process is in a message $a \triangleleft l : v$, then one cannot infer the type of *all* the messages an object located at a may receive.

This justifies the introduction of the *kinded type systems* $\mathrm{TA}_{[]\langle\rangle}$ and $\mathrm{TA}_{[]\mu\langle\rangle}$, where type-variables are assigned *kinds* describing constraints on the types that may replace the type-variables [**43**]. In the example above, to name a one could assign some type-variable t, and constrain the types that may substitute t to have *at least* a component $l \colon \alpha$, for α the type of v. This refinement yields a practical notion of typing subsumption and principal typings. The main properties of the kinded systems, in addition to those of $\mathrm{TA}_{[]}$ and $\mathrm{TA}_{[]\mu\langle\rangle}$, are the following:

4. Correctness with respect to $\mathrm{TA}_{[]}$ (and to $\mathrm{TA}_{[]\mu\langle\rangle}$)
   A term is typable in $\mathrm{TA}_{[]\langle\rangle}$ ($\mathrm{TA}_{[]\mu\langle\rangle}$) if and only if it is typable in $\mathrm{TA}_{[]}$ ($\mathrm{TA}_{[]\mu}$).
5. Existence and computability of principal typings (PT)
   If P is typable, there exists a principal typing for P in $\mathrm{TA}_{[]\langle\rangle}$; it can be computed from P.

The main properties of the type systems discussed in this thesis are summarized in the table below.

	$\mathrm{TA}_{[]}$	$\mathrm{TA}_{[]\langle\rangle}$	$\mathrm{TA}_{[]\mu}$	$\mathrm{TA}_{[]\mu\langle\rangle}$	$\mathrm{TA}_{()\mu}$	$\mathrm{TA}_{\rightarrow}$	$\mathrm{TA}_{\rightarrow\mu}$
SR	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
$\vdash P \blacktriangleright \Gamma?$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
$\vdash P \blacktriangleright ?$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
PT		$\surd$		$\surd$	$\surd$	$\surd$	$\surd$
SN						$\surd$	

Three systems where included for reference. System $\mathrm{TA}_{()\mu}$ is the type system for the polyadic π-calculus developed by Honda and myself [**64, 65**]. System $\mathrm{TA}_{\rightarrow}$

is the simple type system for the λ-calculus, and system $\mathrm{TA}_{\to\mu}$ is the recursive type system for the λ-calculus developed by Cardone and Coppo [**15, 16**].

The new acronym, SN, stands for the strong normalization property of the λ-calculus that says that typable terms are strongly normalizable. There is no notion of strong normalization for process calculi; SN was only introduced to emphasize that although one of the most important properties of system $\mathrm{TA}_{\to}$ is lost in system $\mathrm{TA}_{\to\mu}$, there seems to be nothing to lose when going from system $\mathrm{TA}_{\square}$ to system $\mathrm{TA}_{\square\mu}$, except perhaps some simplicity at the syntactic level.

Kinded Unification. Unification of kinded infinite trees is crucial in proving the existence and computability of principal typings in the recursive kinded system $\mathrm{TA}_{\square\mu\langle\rangle}$, which in turn forms the basis of the proof of decidability of typing assignment and computability of typing inference in the recursive type system $\mathrm{TA}_{\square\mu}$.

Because the results on unification of kinded infinite trees can be used in different contexts, I made the chapter as general as possible, and independent of the rest of the thesis. Possible applications for the results in this chapter include inference in richer type systems containing, for example, a function space constructor [**13, 43**] or other type constructors [**56**]; and resolution in constraint logic programming with records [**54**]. Specifically I show that:

1. If a kinded set of equations over infinite trees is unifiable, it has a most general unifier using *at most* the variables in the equations and in the kinding; and
2. If, in addition, the kinded set of equations is regular (has only a finite number of subtrees), its most general unifier is also regular and can be effectively computed.

I present a unification algorithm for extracting the most general unifier of a kinded set of equations, or to detect that the kinded set of equations is not unifiable, thus extending Huet's algorithm [**33**] into the kinded setting, and Ohori's algorithm [**43**] into the infinite setting.

3. How to Read This Thesis

This thesis may be read in a number of different ways by readers with different interests.

1. An experienced reader looking for the main ideas and results of the type systems should cover chapters 3 and 4, after having browsed chapter 2 to grasp the fundamentals of the calculus.
2. A reader interested in a name-passing calculus with labels may start reading the next chapter, and, if also interested in type systems, continue through the subsequent chapters.
3. A reader who wants to study the problem of unification of kinded infinite trees may proceed directly to chapter 6, since this chapter is independent of the rest of the thesis. To get quickly to the unification algorithm, she should skim section 1 for the notions of records and kindings, section 3 for the notion of closure of a kinding, and then go through section 4 where the algorithm is presented.
4. A reader interested in type systems for the π- or the ν-calculus should cover chapters 3 and 4 of the simple and the recursive system, while trying to imagine

   terms $a \triangleleft l : \tilde{v}$ replaced by $\overline{a}\tilde{v}.\mathbf{0}$,
   terms $a \triangleright [l_1 : (\tilde{x}_1)\, P_1 \,\&\, \cdots \,\&\, l_n : (\tilde{x}_n)\, P_n]$ replaced by $a(\tilde{x}).P$, and
   types $[l_1 \colon \tilde{\alpha}_1, \ldots, l_n \colon \tilde{\alpha}_n]$ replaced by $(\tilde{\alpha})$.

   This yields the (polyadic) ν-calculus. The π-calculus is only a step ahead. All the properties (and proofs) discussed in these two chapters hold in any of the two calculi. For principal-types, she may proceed directly to section 5 of chapter 5 where the inference algorithm is presented. The algorithm holds for the π-calculus with few changes. The proof of the correction of the algorithm easily adapts to any of the two calculi.

CHAPTER 2

The Calculus of Objects

This chapter introduces the calculus of objects, its syntax and semantics. The semantics, however, will only be developed as far as needed to prove the subject-reduction property in the type systems to be developed in the subsequent chapters. Since the goal of this thesis is the study of type structures that arise in a name-passing calculus with labels, the behaviour of programs will not be explored.

1. Syntax

Finite sequences, or strings, will be used for different purposes, including: sequences of names in the syntax of the calculus, sequences of types in the syntax of types, and sequences of positive integers in the domain of infinite trees.

DEFINITION 2.1 (Finite sequences). Let A be a set. The set of *finite sequences* (or strings) of symbols in A is denoted A^*. The empty sequence is ε, and, when n is not important, the sequence $a_1 a_2 \cdots a_n$ in A^* is abbreviated to $\tilde{a}$. The length of a sequence $\tilde{a}$, notation $\text{length}(\tilde{a})$, is defined as $\text{length}(\varepsilon) = 0$, and $\text{length}(a_1 a_2 \cdots a_n) = n$ for $n > 0$. The set of symbols in $\tilde{a}$, notation $\{\tilde{a}\}$, is defined as $\{\varepsilon\} = \emptyset$, and $\{a_1 a_2 \cdots a_n\} = \{a_1, a_2, \ldots, a_n\}$ for $n > 0$.

DEFINITION 2.2 (Syntax). Assume an infinite set $\mathbf{N}$ of *names*, and a set $\mathbf{L}$ of *labels*. Possibly subscripted or primed, lower case Roman letters will denote names, while the particular letter l will denote a label. The set $\mathbf{P}$ of *terms* or *processes* is defined inductively by the grammar

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

for $n \geq 0$, where upper case Roman letters $P, Q, \ldots$ denote terms. Names in each $\tilde{x}_i$, as well as labels $l_1, \ldots, l_n$, are assumed to be pairwise distinct. Words 'term' and 'process' will be used interchangeably.

Terms of the form $a \triangleleft l : \tilde{v}$ are called *messages*; name a is the *target*, and $l : \tilde{v}$ is the *communication* of the message. Label l selects one particular method in the target object while the sequence of names $\tilde{v}$ constitutes the actual contents of the message. *Objects* are terms of the form $a \triangleright M$ where a is the *location* (or identifier or address) of the object, and M represents a (finite, possibly empty) unordered collection of *methods*. Each method, of the form $l : (\tilde{x})\, P$, is labelled by a different label l. The sequence of names $\tilde{x}$ represents the formal parameters, while term P represents the body of the method. Methods are essentially unordered, so that $[l : (\tilde{x})\, P \,\&\, l' : (\tilde{y})\, Q]$ and $[l' : (\tilde{y})\, Q \,\&\, l : (\tilde{x})\, P]$ represent the same collection of methods. I decided not to introduce a separate syntactic category for methods; however I shall reserve letters M and N to denote (finite, possibly empty) collections of methods.

Objects as labelled sums and asynchronous labelled messages are the two distinctive constructors of the calculus. Intuitively, the result of the interaction of a message $a \triangleleft l_i : \tilde{v}$ and an object $a \triangleright [l_1 : (\tilde{x}_1)\, P_1 \,\&\, \cdots \,\&\, l_n : (\tilde{x}_n)\, P_n]$ is the process P_i with $\tilde{x}_i$ replaced by $\tilde{v}$. It is easy to see that such an interaction is well defined only when l_i labels a method in the object, that is when $1 \leq i \leq n$, and when the lengths of $\tilde{v}$ and $\tilde{x}_i$ match.

The remaining constructors are fairly standard in name-passing calculi. The *concurrent composition* of P and Q, notation P, Q, denotes the processes P and Q running in parallel. Terms of the form $\nu x P$ are called *scope restrictions*: they allow creating a new name x visible only in the scope determined by term P. The final constructor of the calculus is *replication* and provides for unbounded computation power. A term of the form $!P$ denotes as many copies of P as needed, all running in parallel. In particular, replication provides for recursive process definition in a simple and mathematically more tractable form than explicit recursion.

REMARK 2.3. In contrast with usual notions of objects, there is not a one-to-one correspondence between objects and names: there may be more than one object sharing the same location, as there may be names without an associated object. Also, there is no distinguished name "self"; this can be just another name, location of some object. Finally, objects are usually persistent, in the sense that they survive the reception and the processing of messages. In the calculus here proposed, for an object $a \rhd [l_1 : (\tilde{x}_1)\, P_1 \,\&\cdots\&\, l_n : (\tilde{x}_n)\, P_n]$ to survive the reception of messages, each method-body P_i must explicitly instantiate an object located at a. Persistence is achieved by using replication; see example 2.5(4).

NOTATION 2.4. 1. *Inaction*, notation $\mathbf{0}$, is the process that does nothing, and can be thought as an abbreviation for the term $\nu x\, x \rhd []$.

2. Parentheses will be omitted in such a way that, for example, P, Q, R represents $(P, Q), R$.
3. Operators "!" and "ν" bind tighter than ",", so that, for example, $\nu P, Q$ denotes the term $(\nu P), Q$, and $!P, Q$ denotes the term $(!P), Q$.
4. When the contents of a message is the empty sequence of names, abbreviate $a \lhd l : \varepsilon$ to $a \lhd l$. Also abbreviate method $l : (\varepsilon)\, P$ to $l : P$.
5. Multiple name-restriction $\nu x_1 \cdots \nu x_n P$ is abbreviated to $\nu x_1 \cdots x_n P$, or to $\nu \tilde{x} P$ when n is not important. In this way $\nu \varepsilon P$ represents the process P.

EXAMPLES 2.5. (1) Forwarder of *put* messages. Term

$$a \rhd [put : (x)\, b \lhd put : x]$$

represents an object receiving a *put*-labelled message at a, and sending it to b.
(2) Suppose $\mathbf{L}$ is finite. If $\mathbf{L} = \{l_1, \ldots, l_n\}$, term

$$a \rhd [l_1 : (x)\, b \lhd l_1 : x \;\&\; \cdots \;\&\, l_n : (x)\, b \lhd l_n : x]$$

represents a generic forwarder, forwarding a monadic message from a to b. Notice that generic forwarders are only possible under the assumption of the finiteness of $\mathbf{L}$. To forward an arbitrary sequence of names, one forwards a single name representing a list of names (see 6 below).
(3) Boolean values. Term

$$t \rhd [val : (r)\, r \lhd true]$$

represents an object that replies *true* to a message *val*, and the object

$$f \rhd [val : (r)\, r \lhd false]$$

replies *false* to the same message. If the above terms represent an encoding for the (ephemeral) boolean values, then term

$$\nu x(b \lhd \mathit{val}\colon x, x \rhd [\mathit{true}\colon P \mathbin{\&} \mathit{false}\colon Q]) \qquad (x \text{ fresh})$$

stands for a conditional term

$$\textbf{if } b \textbf{ then } P \textbf{ else } Q.$$

(4) Recursive definitions. Assume a set of constants, and let letter D denote an element of this set. Introduce a derived constructor, $D\tilde{v}$, to represent the application of a sequence of names $\tilde{v}$ to constant D. When a constant application is used in some term, assume an equation of the form

$$D \quad \stackrel{\text{def}}{=} \quad (\tilde{x})\, P$$

has been defined. To allow expressing explicit recursion, allow D to occur in P. Now, occurrences of $D\tilde{v}$ in terms (not in equations) are translated into terms of the form

$$\nu u(u \lhd \mathit{recur}\colon \tilde{v}, !u \rhd [\mathit{recur}\colon (\tilde{x})\, P']) \qquad (u \text{ fresh})$$

where P' is obtained from P by replacing occurrences of $D\tilde{z}$ by $u \lhd \mathit{recur}\colon \tilde{z}$. (If D does not occur in P, one may simply translate $D\tilde{v}$ into $P\{\tilde{v}/\tilde{x}\}$, that is into P with names $\tilde{x}$ replaced by names $\tilde{v}$.)

(5) Mutable buffer cell. The term $\mathrm{Cell}\, sv$, with

$$\mathrm{Cell} \quad \stackrel{\text{def}}{=} \quad (sv)\, s \rhd [\mathit{read}\colon (r)\, r \lhd \mathit{val}\colon v, \mathrm{Cell}\, sv \mathbin{\&} \mathit{write}\colon (u)\, \mathrm{Cell}\, su],$$

represents a cell offering *read* and *write* operations. With a *read* message comes a name intended to receive a *val*-labelled message with the contents of the cell; with a *write* message comes the new value for the cell. According to the encoding of recursion, the expansion of $\mathrm{Cell}\, sv$ is the term

$$\begin{aligned}&\nu x(x \lhd \mathit{recur}\colon sv,\\ &\quad !x \rhd [\mathit{recur}\colon (sv)\, s \rhd [\mathit{read}\colon (r)\, r \lhd \mathit{val}\colon v, x \lhd \mathit{recur}\colon sv \mathbin{\&} \mathit{write}\colon (u)\, x \lhd \mathit{recur}\colon su]]).\end{aligned}$$

(6) Lists. Similarly to boolean values, lists can be built from two constructors—Nil and Cons—defined as follows.

$$\begin{aligned}\text{Nil} &\stackrel{\text{def}}{=} (n)\, n \rhd [\mathit{val}\!:\!(r)\, r \lhd \mathit{nil}]\\ \text{Cons} &\stackrel{\text{def}}{=} (c f s)\, c \rhd [\mathit{val}\!:\!(x)\, x \lhd \mathit{cons}\!:\! f s]\end{aligned}$$

To messages *val*, Nil cells reply *nil* as if saying "I am a Nil cell;" Cons cells reply $\mathit{cons} : f s$ saying "I am a Cons cell and here is my first and second elements." Actual lists can be built from these two constructors 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

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

2. Name Substitution

Methods and scope restriction are the *name binding operators* of the calculus. An occurrence of a name x in a term is *bound* if it occurs in the part P of a method $l : (\tilde{w} x \tilde{y})\, P$ or of a term $\nu x P$; otherwise it is called *free*. A name may occur both free and bound in a term: take for example u in term $u \rhd [l : (u)\, \mathbf{0}]$. If a name has at least one free occurrence in P, it is called a *free name of P*.

Definition 2.6 (Free names). The set of free names in a term P, notation $\text{fn}(P)$, can be inductively defined as follows.

$$\begin{aligned}\text{fn}(a \lhd l\!:\!\tilde{v}) &= \{a\} \cup \{\tilde{v}\};\\ \text{fn}(a \rhd [l_1 : (\tilde{x}_1)\, P_1 \,\&\, \cdots \,\&\, l_n : (\tilde{x}_n)\, P_n]) &= \{a\} \cup \text{fn}(P_1) \setminus \{\tilde{x}_1\} \cup \cdots \cup \text{fn}(P_n) \setminus \{\tilde{x}_n\};\\ \text{fn}(P, Q) &= \text{fn}(P) \cup \text{fn}(Q);\\ \text{fn}(\nu x P) &= \text{fn}(P) \setminus \{x\};\\ \text{fn}(!P) &= \text{fn}(P).\end{aligned}$$

To simplify the treatment of substitution and some proofs, I follow the below convention on names (cf. the variable convention in Barendregt [**5**]).

Name Convention 2.7. Bound names in terms appearing in mathematical contexts (but not necessarily in examples) are chosen to be distinct from the free names.

Definition 2.8 (Name substitution). The result of substituting z for x in a sequence of names $\tilde{u}$, notation $\tilde{u}[z/x]$, is inductively defined as follows.

$$\begin{aligned}
\varepsilon[z/x] &= \varepsilon; \\
(z\tilde{u})[z/x] &= x(\tilde{u}[z/x]); \\
(y\tilde{u})[z/x] &= y(\tilde{u}[z/x]) \qquad \text{if } y \neq z.
\end{aligned}$$

The result of substituting z for the free occurrences of x in a term P, notation $P[z/x]$, or $P\sigma$ when $\sigma = [z/x]$, is inductively defined as follows.

$$\begin{aligned}
(a \lhd l : \tilde{v})\sigma &= a\sigma \lhd l : \tilde{v}\sigma; \\
(a \rhd [l_1 : (\tilde{x}_1)\, P_1 \,\&\, \cdots \,\&\, l_n : (\tilde{x}_n)\, P_n])\sigma &= a\sigma \rhd [l_1 : (\tilde{x}_1)\, P_1\sigma \,\&\, \cdots \,\&\, l_n : (\tilde{x}_n)\, P_n\sigma]; \\
(P, Q)\sigma &= P\sigma, Q\sigma; \\
(\nu y P)\sigma &= \nu y P\sigma; \\
(!P)\sigma &= !P\sigma.
\end{aligned}$$

The result of substituting $\tilde{z}$ for the free occurrences of $\tilde{x}$ in a term P, notation $P\{\tilde{z}/\tilde{x}\}$, is defined as follows.

$$\begin{aligned}
P\{\varepsilon/\varepsilon\} &= P; \\
P\{z\tilde{z}/x\tilde{x}\} &= (P[z/x])\{\tilde{z}/\tilde{x}\}.
\end{aligned}$$

Notice that $P\{\tilde{z}/\tilde{x}\}$ is defined only when $\text{length}(\tilde{z}) = \text{length}(\tilde{x})$.

Definition 2.9 (Alpha-congruence).

1. *Change of bound names* in P is the replacement of a part $\nu x Q$ or $(\tilde{w}x\tilde{y})\, Q$ by $\nu z Q[z/x]$ or $(\tilde{w}z\tilde{y})\, Q[z/x]$, respectively, where z does not occur (at all) in Q.
2. P is *α-congruent* with Q, notation $P \equiv_\alpha Q$, if Q results from P by a finite (possibly empty) series of changes of bound names.

3. Reduction

The central notion of reduction is the interaction between a message and an object. Messages and objects may occur in terms far apart, and it is necessary to bring messages and objects into syntactically contiguous positions before reduction can happen. Two notions help in simplifying the formulation of reduction, separating the static, syntax-related aspects, from the dynamic, computation-related aspects of the calculus; they are: *structural congruence* and *normal forms* of processes. Together, these two notions allow defining reduction with a simple pair of rules.

DEFINITION 2.10 (Structural congruence). *Structural congruence*, notation $P \equiv Q$, is the smallest congruence relation over $\mathbf{P}$, satisfying the following rules.

1. $P \equiv Q$ if $P \equiv_\alpha Q$;
2. $P, Q \equiv Q, P$ and $(P, Q), R \equiv P, (Q, R)$;
3. $a \triangleright M \equiv a \triangleright N$ if N is permutation of M;
4. $\nu x P, Q \equiv \nu x(P, Q)$ and $\nu x Q \equiv Q$, when $x \notin \operatorname{fn} Q$;
5. $!P \equiv P, !P$.

DEFINITION 2.11 (Base terms). Objects (replicated or not) and messages are called *base-terms*. The concurrent composition of an arbitrary number of base terms is denoted by $\tilde{P}$.

The notion of normal forms over terms is introduced with the sole purpose of simplifying the formulation of the reduction relation, and should not be confused with that in the λ-calculus.

PROPOSITION 2.12 (Normal form). *Every term in* $\mathbf{P}$ *can be transformed into a structural congruent term of the form* $\nu\tilde{x}\tilde{P}$.

PROOF. By induction on the structure of terms. If the term is a base-term, the proof is done. Otherwise suppose P is base-term; $\nu x P$ is obviously a base-term. Now suppose $\nu\tilde{x}\tilde{P}$ and $\nu\tilde{y}\tilde{Q}$ are two base-terms. Let $\nu\tilde{x}'\tilde{P}' \equiv_\alpha \nu\tilde{x}\tilde{P}$ such that $\{\tilde{x}'\} \cap \operatorname{fn}\tilde{Q} = \emptyset$. Then $\nu\tilde{x}'\tilde{P}', \nu\tilde{y}\tilde{Q} \equiv^* \nu\tilde{x}'(\tilde{P}', \nu\tilde{y}\tilde{Q}) \equiv \nu\tilde{x}'(\nu\tilde{y}\tilde{Q}, \tilde{P}')$. Similarly let $\nu\tilde{y}'Q' \equiv_\alpha \nu\tilde{y}\tilde{Q}$ such that $\{\tilde{y}'\} \cap \operatorname{fn}\tilde{P} = \emptyset$. Then $\nu\tilde{x}'(\nu\tilde{y}'\tilde{Q}', \tilde{P}') \equiv^* \nu\tilde{x}'\nu\tilde{y}'(\tilde{Q}', \tilde{P}')$. $\square$

The reception of a message by an object constitutes the basic communication mechanism of the calculus, and encapsulates a sequence of simple operations in a single atomic operation called *message application*.

DEFINITION 2.13 (Message application). Let C be the communication $l_i : \tilde{v}$ of some message, and let M be the collection of methods $[l_1 : (\tilde{x}_1)\, P_1 \,\&\cdots\&\, l_n : (\tilde{x}_n)\, P_n]$ of some object. The application of C to M, notation $M \bullet C$, is the term $P\{\tilde{v}/\tilde{x}_i\}$, provided that $1 \le i \le n$ and that $\operatorname{length}\tilde{v} = \operatorname{length}\tilde{x}_i$.

Explicating, $[l_1 : (\tilde{x}_1)\, P_1 \,\&\cdots\&\, l_n : (\tilde{x}_n)\, P_n] \bullet l_i : \tilde{v}$ involves the following steps.

1. The selection of the method labelled with l_i;
2. the substitution of $\tilde{x}_i$ by $\tilde{v}$ in P_i, to obtain $P\{\tilde{v}/\tilde{x}_i\}$;
3. the launching of process $P_i\{\tilde{v}/\tilde{x}_i\}$, discarding all other methods.

DEFINITION 2.14 (Reduction). *One-step reduction*, notation $P \to Q$, is the smallest relation generated by the following rules.

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

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

Multi-step reduction, or simply *reduction*, notation $P \twoheadrightarrow Q$, is the relation $\equiv \cup \to^*$, where $\to^*$ is the transitive closure of $\to$.

Notice that there is no automatic rewriting of the object involved in a communication: for an object $a \triangleright M$ to survive a message application, a term of the form $a \triangleright M'$ must occur in $M \bullet C$, or else replication must be used (see example 3 below).

EXAMPLES 2.15. (1) One has the following reductions.

$$a \triangleright [put : (x)\, b \triangleleft put : x],\; a \triangleleft put : v \;\to\; b \triangleleft put : v$$

$$a \triangleright [put : (x)\, b \triangleleft put : x \;\&\; get : (x)\, b \triangleleft get : x],\; a \triangleleft put : v \;\to\; b \triangleleft put : v$$

(2) Noting that x is fresh, one has the following deduction.

$$
\begin{aligned}
&\textbf{if}\, b \,\textbf{then}\, P \,\textbf{else}\, Q,\ \mathrm{True}\, b \overset{\text{def}}{=} \\
&\nu x(b \lhd \mathsf{val}{:}\, x, x \rhd [\mathit{true}{:}\, P \,\&\, \mathit{false}{:}\, Q]),\ b \rhd [\mathsf{val}{:}\, (r)\, r \lhd \mathit{true}] \ \equiv^* \\
&\nu x(b \rhd [\mathsf{val}{:}\, (r)\, r \lhd \mathit{true}],\ b \lhd \mathsf{val}{:}\, x,\ x \rhd [\mathit{true}{:}\, P \,\&\, \mathit{false}{:}\, Q]) \ \to \\
&\nu x(x \lhd \mathit{true},\ x \rhd [\mathit{true}{:}\, P \,\&\, \mathit{false}{:}\, Q]) \ \equiv \\
&\nu x(x \rhd [\mathit{true}{:}\, P \,\&\, \mathit{false}{:}\, Q],\ x \lhd \mathit{true}) \ \to \\
&\nu x P \ \equiv \\
&P
\end{aligned}
$$

Similarly, one has the following reduction.

$$\textbf{if}\, b \,\textbf{then}\, P \,\textbf{else}\, Q,\ \mathrm{False}\, b \ \equiv^* \to \equiv \to \equiv \ Q$$

(3) Write P for the term

$$x \rhd [\mathit{recur}{:}\, (sv)\, s \rhd [\mathit{read}{:}\, (r)\, r \lhd \mathsf{val}{:}\, v,\ x \lhd \mathit{recur}{:}\, sv \,\&\, \mathit{write}{:}\, (u)\, x \lhd \mathit{recur}{:}\, su]].$$

One has the following reduction.

$$
\begin{aligned}
&\mathrm{Cell}\, sv,\ s \lhd \mathit{read}{:}\, r \overset{\text{def}}{=} \\
&\nu x(x \lhd \mathit{recur}{:}\, sv,\ !P),\ s \lhd \mathit{read}{:}\, r \ \equiv^* \\
&\nu x(P,\ x \lhd \mathit{recur}{:}\, sv,\ !P,\ s \lhd \mathit{read}{:}\, r) \ \to \\
&\nu x(s \rhd [\mathit{read}{:}\, (r)\, r \lhd \mathsf{val}{:}\, v,\ x \lhd \mathit{recur}{:}\, sv \,\&\, \mathit{write}{:}\, (u)\, x \lhd \mathit{recur}{:}\, su],\ !P,\ s \lhd \mathit{read}{:}\, r) \ \equiv^* \\
&\nu x(s \rhd [\mathit{read}{:}\, (r)\, r \lhd \mathsf{val}{:}\, v,\ x \lhd \mathit{recur}{:}\, sv \,\&\, \mathit{write}{:}\, (u)\, x \lhd \mathit{recur}{:}\, su],\ s \lhd \mathit{read}{:}\, r,\ !P) \ \to \\
&\nu x(r \lhd \mathsf{val}{:}\, v,\ x \lhd \mathit{recur}{:}\, sv,\ !P) \ \equiv^* \\
&\nu x(x \lhd \mathit{recur}{:}\, sv,\ !P),\ r \lhd \mathsf{val}{:}\, v \overset{\text{def}}{=} \\
&\mathrm{Cell}\, sv,\ r \lhd \mathsf{val}{:}\, v
\end{aligned}
$$

Similarly, one has the following reduction.

$$\mathrm{Cell}\, sv,\ s \lhd \mathit{write}{:}\, u \ \equiv^* \to \equiv^* \to \equiv^* \ \mathrm{Cell}\, su$$

4. Notes

The idea of extracting a congruence relation from reduction is due to Milner [**37**], inspired by the structural equivalence of Berry and Boudol's Chemical Abstract Machine [**10, 11**]. Notice however that in the latter, the structural equivalence is not a congruence relation. There seems to be no general agreement on the rules in the definition of structural congruence. Sangiorgi mentions some guidelines, including that the rules should provide for some intuition on the operators; should allow a simple formulation of reduction; should be needed in proofs; and there should be only a few of them [**53**]. This was ultimately what motivated my selection of rules, among the vast array of possibilities.

Replication in process-calculi was introduced by Milner [**37**], inspired by the "of course" operator of linear logic [**23**]. The original proposal of the π-calculus used equations between processes to provide for explicit recursion, similarly to what I did in example 2.5(4). Replication provides a simpler tool to define infinite behaviour, dispensing equations and constants over terms; Milner showed that explicit recursion can be transformed into replication [**37**] in much the same way as in example 2.5(4).

I proposed a formulation of the π-calculus that uses variables over agents (as opposed to constants in example 2.5(4)) and brings the equations directly into the calculus, thus providing for a clean treatment of recursion, while avoiding replication [**61**]. Although the sole purpose of that work was to allow a form of Damas-Milner predicative polymorphism [**19**], the same technique can be used to obtain a calculus of objects with explicit recursion [**63**].

The choice of incorporating the replication rule in structural congruence has its opponents. Honda and Yoshida [**30**] prefer to place the treatment of replication directly in the definition of reduction, on the grounds that the rule for replication increases the size of terms, in contrast with the remaining rules in structural congruence.

Inaction as a derived constructor is not usual in process-calculi. Semantics considerations may force to place it directly in the syntax of processes, but type assignment seems not to require inaction as primitive. Observe that this decision forces the introduction of an also unusual structural congruence rule '$\nu x P \equiv P$ if

$x \notin \mathrm{fn}\, P$' in order to get rid of useless scope restrictions, which would otherwise be accounted for by the more conventional rule '$\nu x \mathbf{0} \equiv \mathbf{0}$'. Either derived or primitive, $\mathbf{0}$ is a fairly easy process to type. Also, all the structural congruence rules I have encountered so far seem to verify the congruence lemma 3.17, so that for typing purposes it does not make much difference what structural congruence rules one chooses.

Finally, normal forms for processes in view of the simplification of the statement of reduction were first used by Honda [**29**].

CHAPTER 3

Simple Typing Assignment

This chapter introduces a notion of types for names, and a formal system assigning typings, that is, sets of name-type pairs, to terms. A series of properties of the type system are discussed, culminating with subject-reduction, decidability of typing assignment, and computability of typing inference.

1. Simple Types

DEFINITION 3.1 (Simple types). Assume an infinite set $\mathbf{V}$ of *type-variables*, and let $\mathbf{L}$ be the set of labels introduced in definition 2.2. Further assume that $\mathbf{V}$ and $\mathbf{L}$ are disjoint. Letter t will be used for type-variables. The set $\mathbf{T}_{[]}$ of *simple types* is inductively defined by the grammar

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

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 $n \geq 0$. Lower case Greek letters $\alpha, \beta, \ldots$ will denote types.

Types of the form $[l_1 \colon \tilde{\alpha}_1, \ldots, l_n \colon \tilde{\alpha}_n]$ are called *records*. Parts $l_i \colon \tilde{\alpha}_i$, for $i = 1, \ldots, n$, are the *components* of the record. A component $l \colon \varepsilon$, for ε the empty sequence of types, is abbreviated to l.

Intuitively, a type of the form $[l_1 \colon \tilde{\alpha}_1, \ldots, l_n \colon \tilde{\alpha}_n]$ denotes the 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 in $\tilde{\alpha}_i$. For simplicity, I do not consider constant types; their incorporation should be straightforward.

REMARK 3.2. Similarly to methods, records are unordered, so that the record $[l \colon \tilde{\alpha}, l' \colon \tilde{\beta}]$ and the record $[l' \colon \tilde{\beta}, l \colon \tilde{\alpha}]$ represent the same type. A record α can be seen as the equivalence class of all permutations of the components of α. Alternatively, records can be seen as partial maps of finite domain from the set of labels $\mathbf{L}$ into sequences of types in $\mathbf{T}_{[]}^*$.

2. Typings

DEFINITION 3.3 (Typings).

1. *Type assignment* is any expression

$$x\colon \alpha,$$

   for all names x and types α, where x is called the *subject*, and α the *predicate* of the assignment.
2. *Typings* are finite 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. Upper case Greek letters $\Gamma, \Delta, \ldots$ will denote typings.
3. Let $\Gamma = \{x_1\colon \alpha_1, \ldots, x_n\colon \alpha_n\}$ be a typing. Thinking of typings as partial maps of finite domain, define $\operatorname{dom}\Gamma = \{x_1, \ldots, x_n\}$, and $\Gamma x_i = \alpha_i$ for $1 \leq i \leq n$.
4. Let $N \in \operatorname{dom}\Gamma$ be a set of names, and Γ a typing. Then, the *restriction of* Γ *to* N is the typing

$$\Gamma\restriction N = \{x\colon \Gamma x \mid x \in N\}.$$

5. Let x be a name, and Γ a typing. Then, Γ *reduced by* x is the typing

$$\Gamma/x = \Gamma\restriction(\operatorname{dom}\Gamma \setminus \{x\}).$$

NOTATION 3.4. Let $\tilde{x} = x_1 \cdots x_n$ be a sequence of pairwise distinct names, and $\tilde{\alpha} = \alpha_1 \cdots \alpha_n$ a sequence of types, for some $n \geq 0$. Then,

1. $\tilde{x}\colon \tilde{\alpha}$ denotes the typing $\{x_1\colon \alpha_1, \ldots, x_n\colon \alpha_n\}$;
2. $\Gamma \cdot \tilde{x}\colon \tilde{\alpha}$ denotes the typing $\Gamma \cup \tilde{x}\colon \tilde{\alpha}$, provided $\{\tilde{x}\} \cap \operatorname{dom}\Gamma = \emptyset$.

Compatible types can be combined by a simple set-union operation.

DEFINITION 3.5 (Typing compatibility).

1. Typings Γ and Δ are *compatible*, notation $\Gamma \asymp \Delta$, if $\Gamma x = \Delta x$ for all $x \in \operatorname{dom}\Gamma \cap \operatorname{dom}\Delta$.
2. Typings $\Gamma_1, \ldots, \Gamma_n$ are *pairwise compatible*, notation $\Gamma_1 \asymp \cdots \asymp \Gamma_n$, if $\Gamma_i \asymp \Gamma_j$ for all $1 \le i, j \le n$.

PROPOSITION 3.6. *Let $\Gamma \asymp \Delta \asymp \Theta$. Then*
1. $\Gamma \cup \Delta$ *is a typing;*
2. $\Gamma \asymp \Theta$ *and* $\Gamma \cup \Delta \asymp \Theta$*;*
3. *If* $\Gamma' \subseteq \Gamma$*, then* $\Gamma' \asymp \Delta$.

PROOF. Directly from the definition of typing compatibility. □

DEFINITION 3.7 (Typing substitution). The result of substituting z for x in Γ, notation $\Gamma[z/x]$, is defined as follows.

$$\begin{array}{rcll} (\Gamma \cdot x\colon \alpha)[z/x] & = & \Gamma \cup \{z\colon \alpha\} & \text{if } z \notin \operatorname{dom}\Gamma \text{ or } \Gamma z = \alpha; \\ \Gamma[z/x] & = & \Gamma & \text{if } x \notin \operatorname{dom}\Gamma; \\ \Gamma[z/x] & & \text{is undefined otherwise.} & \end{array}$$

Notice that $\Gamma[z/x]$ is not defined when both x and z are subjects in Γ and $\Gamma x \neq \Gamma z$. When defined, it is easy to see that $\Gamma[z/x]$ is a typing. I write $\Gamma[z/x] \asymp \Delta[z/x]$ to mean both that $\Gamma[z/x]$ and $\Delta[z/x]$ are defined, and that $\Gamma[z/x] \asymp \Delta[z/x]$.

Typing compatibility is preserved by substitution.

PROPOSITION 3.8. *If $\Gamma \asymp \Delta$, then $\Gamma[z/x] \asymp \Delta[z/x]$.*

PROOF. Directly from the definitions of typing compatibility and substitution. □

3. Simple Typing Assignment System $\mathbf{TA}_{[]}$

DEFINITION 3.9 (Simple typing assignment system $\mathrm{TA}_{[]}$). The simple typing assignment system $\mathrm{TA}_{[]}$ is a formal theory in the sense of Hindley and Seldin [**26**]. Its *formulas* are expressions

$$P \blacktriangleright \Gamma,$$

for all terms P and typings Γ. Its *axioms* are

(MSG) $\quad a \triangleleft l_1 : \tilde{v} \blacktriangleright \{\tilde{v}\colon \tilde{\alpha}_1, a\colon [l_1\colon \tilde{\alpha}_1, \ldots, l_n\colon \tilde{\alpha}_n]\} \qquad (n \ge 1)$

and the particular case of the OBJ-rule when $n = 0$. Its *rules* are the following:

(OBJ) (condition: $\{a\colon [l_1\colon \tilde\alpha_1, \dots, l_n\colon \tilde\alpha_n]\} \asymp \Gamma_1 \asymp \cdots \asymp \Gamma_n$)

$$\frac{P_1 \blacktriangleright \Gamma_1 \cdot \tilde x_1\colon \tilde\alpha_1 \qquad \cdots \qquad P_n \blacktriangleright \Gamma_n \cdot \tilde x_n\colon \tilde\alpha_n \qquad (n \geq 0)}{a \triangleright [l_1\colon(\tilde x_1)\, P_1 \,\&\, \cdots \,\&\, l_n\colon(\tilde x_n)\, P_n] \blacktriangleright \{a\colon [l_1\colon \tilde\alpha_1, \dots, l_n\colon \tilde\alpha_n]\} \cup \Gamma_1 \cup \cdots \cup \Gamma_n}$$

(SCOP)
$$\frac{P \blacktriangleright \Gamma}{\nu x P \blacktriangleright \Gamma / x}$$

(COMP)
$$\frac{P \blacktriangleright \Gamma \qquad Q \blacktriangleright \Delta}{P, Q \blacktriangleright \Gamma \cup \Delta} \qquad \text{(condition: } \Gamma \asymp \Delta\text{)}$$

(REPL)
$$\frac{P \blacktriangleright \Gamma}{!P \blacktriangleright \Gamma}$$

(WEAK)
$$\frac{P \blacktriangleright \Gamma}{P \blacktriangleright \Gamma \cup \{x\colon \alpha\}} \qquad \text{(condition: } x \notin \operatorname{dom}\Gamma\text{)}$$

Write $\vdash_{[]} P \blacktriangleright \Gamma$ if there is a *deduction* of $P \blacktriangleright \Gamma$, that is, if there is a tree of formulas, with those at the top branches being an axiom, the others being deduced from those immediately above them by a rule, and the bottom being $P \blacktriangleright \Gamma$. In this case say P is *typable*, and call Γ a *well-typing* for P. If Γ is empty, simply write $\vdash_{[]} P$.

EXAMPLES 3.10. (1) Call Bool to the type $[\mathit{val}\colon [\mathit{true}, \mathit{false}]]$. The following is an admissible rule of system $\mathrm{TA}_{[]}$.

$$\frac{P \blacktriangleright \Gamma \qquad\qquad Q \blacktriangleright \Delta}{\textbf{if } b \textbf{ then } P \textbf{ else } Q \blacktriangleright \{b\colon \mathrm{Bool}\} \cup \Gamma \cup \Delta} \qquad \text{(condition: } \{b\colon \mathrm{Bool}\} \asymp \Gamma \asymp \Delta\text{)}$$

Proof. Call α to the type $[\mathit{true}, \mathit{false}]$. Then one has the following deduction, using the MSG-axiom, and the the OBJ, the COMP, and SCOP-rules. (Notice that x is fresh and as such $x \notin \operatorname{dom}\Gamma \cup \operatorname{dom}\Delta \cup \{b\}$.)

$$\cfrac{\cfrac{\cfrac{P \blacktriangleright \Gamma \qquad\qquad Q \blacktriangleright \Delta}{x \triangleright [\mathit{true}\colon P \,\&\, \mathit{false}\colon Q] \blacktriangleright \{x\colon \alpha\} \cup \Gamma \cup \Delta} \qquad b \triangleleft \mathit{val}\colon x \blacktriangleright \{b\colon \mathrm{Bool}, x\colon \alpha\}}{x \triangleright [\mathit{true}\colon P \,\&\, \mathit{false}\colon Q], b \triangleleft \mathit{val}\colon x \blacktriangleright \{x\colon \alpha, b\colon \mathrm{Bool}\} \cup \Gamma \cup \Delta}}{\textbf{if } b \textbf{ then } P \textbf{ else } Q \blacktriangleright \{b\colon \mathrm{Bool}\} \cup \Gamma \cup \Delta}$$

(2) One has the deduction

$$\frac{b \lhd put{:}x \;\blacktriangleright\; \{b\colon [put\colon t], x\colon t\}}{a \rhd [put{:}(x)\, b \lhd put{:}x] \;\blacktriangleright\; \{a\colon [put\colon t], b\colon [put\colon t]\}}$$

and also

$$\frac{b \lhd put{:}x \;\blacktriangleright\; \{b\colon [put\colon \mathrm{Bool}], x\colon \mathrm{Bool}\}}{a \rhd [put{:}(x)\, b \lhd put{:}x] \;\blacktriangleright\; \{a\colon [put\colon \mathrm{Bool}], b\colon [put\colon \mathrm{Bool}]\}}$$

and also

$$\frac{b \lhd put{:}x \;\blacktriangleright\; \{b\colon [put\colon t, get\colon t'], x\colon t\}}{a \rhd [put{:}(x)\, b \lhd put{:}x] \;\blacktriangleright\; \{a\colon [put\colon t], b\colon [put\colon t, get\colon t']\}}$$

but one does not have

$$\vdash_{[]} a \rhd [put{:}(x)\, b \lhd put{:}x] \;\blacktriangleright\; \{a\colon [put\colon t, get\colon t'], b\colon [put\colon t]\}.$$

(3) One has

$$\vdash_{[]} \mathrm{Cell}\, sv \;\blacktriangleright\; \{s\colon [read\colon [val\colon t], write\colon t], v\colon t\},$$

and also

$$\vdash_{[]} \mathrm{Cell}\, sv \;\blacktriangleright\; \{s\colon [read\colon [val\colon t, get\colon t'], write\colon t], v\colon t\},$$

but not

$$\vdash_{[]} \mathrm{Cell}\, sv \;\blacktriangleright\; \{s\colon [read\colon [val\colon t], write\colon t, get\colon t'], v\colon t\}.$$

(4) Term Cons cvc is not typable in TA$_{[]}$. Recall that Cons cvc is the term

$$c \rhd [val{:}(x)\, x \lhd cons{:}vc].$$

What we have here is a case of a *recurring name-structure*, where c carries a message containing x, and x carries a message containing c. For Cons cvc to be typable, one would need a type α such that $\alpha = [val\colon [cons\colon \beta\alpha, \ldots]]$, for some type β. It is easy to see that no finite type solves the above equation.

REMARK 3.11. The second and the third examples above raise the question as to when can one "specialize" a well-typing and still obtain a well-typing. It seems clear that, if $\vdash_{[]} P \;\blacktriangleright\; \Gamma$, and s is a substitution from type-variables to types, then $\vdash_{[]} P \;\blacktriangleright\; s\Gamma$. The question on when can one add components to record-types is more delicate. The second example above shows that one can add components to the record assigned to name b but not to that of name a. Inspecting the code $a \rhd [put{:}(x)\, b \lhd put{:}x]$ of the forwarder of put messages, one discovers that name a

appears in an object-location position, whereas name b is only "mentioned" in the target of a message. This must ultimately account for the asymmetry of the types assigned to a and b. In fact, one knows exactly every possible type of messages name a may carry: they are $put\colon \alpha$ for all types α. On the other hand, one knows only part of the types of messages name b may carry: they include $put\colon \alpha$ for all types α. It seems fair to conclude that system $\mathrm{TA}_{[]}$ does not possess a simple notion of type specialization (or subsumption), solely based on the substitution of type-variables for types, and on the addition of components to records. This justifies the type system to be introduced in chapter 5.

Interestingly enough, although system $\mathrm{TA}_{[]}$ has no notion of type subsumption, it is decidable whether a given term has a given type, and it is computable whether a given term is typable (theorems 3.24 and 3.25).

Remark 3.12. The fourth example above suggests that terms containing recurring name-structures cannot be typed in $\mathrm{TA}_{[]}$. There is nothing wrong with terms of this form. They are meaningful and quite useful in describing, for example, inductively defined data-structures, such as list, trees, or even natural numbers. Furthermore, they do not cause errors at runtime. Chapter 4 extends system $\mathrm{TA}_{[]}$ to allow typing terms with recurring name-structures.

4. Some Properties of System $\mathbf{TA}_{[]}$

The first property analyses the subjects of well-typings. In particular, well-typings for a given term P assign all the free names in P, and there is a well-typing for P assigning exactly the free names in P. Typings that assign only the free names in a term P, that is, typings Γ such that $\Gamma = \Gamma \restriction \mathrm{fn}\, P$, will be called *$P$-typings.*

Proposition 3.13 (Typing lemma). *Let* $\vdash_{[]} P \blacktriangleright \Gamma$. *Then*

1. $\mathrm{fn}\, P \subseteq \mathrm{dom}\, \Gamma$;
2. $\vdash_{[]} P \blacktriangleright \Gamma \restriction \mathrm{fn}\, P$;
3. *If* $\Gamma' \supseteq \Gamma$, *then* $\vdash_{[]} P \blacktriangleright \Gamma'$;
4. *If* $x \notin \mathrm{fn}\, P$, *then* $\vdash_{[]} P \blacktriangleright \Gamma/x$.

Proof. (1) A simple induction on the deduction of $\vdash_{[]} P \blacktriangleright \Gamma$.

(2) By induction on the deduction of $\vdash_{[]} P \blacktriangleright \Gamma$. When the deduction consists of the MSG-axiom, let $\Gamma = \{a\colon [l_1\colon \tilde{\alpha}_1, \ldots, l_n\colon \tilde{\alpha}_n], \tilde{v}\colon \tilde{\alpha}_1\}$. Then $\vdash_{[]} a \lhd l_1 \colon \tilde{v} \blacktriangleright \Gamma$, and $\Gamma = \Gamma \restriction \{a\tilde{v}\} = \Gamma \restriction \mathrm{fn}(a \lhd l_1 \colon \tilde{v})$.

When the deduction ends with the OBJ-rule, suppose that $\vdash_{[]} P_i \blacktriangleright \Gamma_i$, for $i = 1, \ldots, n$, and that $\{a\colon [l_1\colon \tilde{\alpha}_1, \ldots, l_n\colon \tilde{\alpha}_n]\} \asymp \Gamma_1 \asymp \cdots \asymp \Gamma_n$. For each i, by induction, $\vdash_{[]} P_i \blacktriangleright \Gamma_i \restriction \mathrm{fn}\, P_i$. Use the WEAK-rule as many times as needed to obtain $\vdash_{[]} P_i \blacktriangleright (\Gamma_i \restriction \mathrm{fn}\, P_i) \cup \{\tilde{x}_i\colon \tilde{\alpha}_i\}$. Obviously $\Gamma_i \restriction \mathrm{fn}\, P_i \subseteq \Gamma_i$ and by lemma 3.6(3), $\{a\colon [l_1\colon \tilde{\alpha}_1, \ldots, l_n\colon \tilde{\alpha}_n]\} \asymp (\Gamma_1 \restriction \mathrm{fn}\, P_1) \asymp \cdots \asymp (\Gamma_n \restriction \mathrm{fn}\, P_n)$. By the OBJ-rule

$$\begin{aligned}
&\vdash_{[]} a \rhd [l_1 \colon (\tilde{x}_1)\, P_1 \,\&\, \cdots \,\&\, l_n \colon (\tilde{x}_n)\, P_n] \blacktriangleright \\
&\{a\colon [l_1\colon \tilde{\alpha}_1, \ldots, l_n\colon \tilde{\alpha}_n]\} \cup (\Gamma_1 \restriction \mathrm{fn}\, P_1) \cup \cdots \cup (\Gamma_n \restriction \mathrm{fn}\, P_n) = \\
&(\{a\colon [l_1\colon \tilde{\alpha}_1, \ldots, l_n\colon \tilde{\alpha}_n]\} \cup \Gamma_1 \cup \cdots \cup \Gamma_n) \restriction \mathrm{fn}(a \rhd [l_1 \colon (\tilde{x}_1)\, P_1 \,\&\, \cdots \,\&\, l_n \colon (\tilde{x}_n)\, P_n]).
\end{aligned}$$

When the deduction ends with the SCOP-rule, suppose that $\vdash_{[]} P \blacktriangleright \Gamma$. By induction $\vdash_{[]} P \blacktriangleright \Gamma \restriction \mathrm{fn}\, P$. By the SCOP-rule $\vdash_{[]} \nu x P \blacktriangleright (\Gamma \restriction \mathrm{fn}\, P)/x = (\Gamma/x) \restriction \mathrm{fn}(\nu x P)$.

When the deduction ends with the WEAK-rule, suppose that $\vdash_{[]} P \blacktriangleright \Gamma$, and that $x \notin \mathrm{dom}\, \Gamma$. By part 1 of this lemma $x \notin \mathrm{fn}\, P$. By induction $\vdash_{[]} P \blacktriangleright \Gamma \restriction \mathrm{fn}\, P = (\Gamma \cup \{x\colon \alpha\}) \restriction \mathrm{fn}\, P$.

The remaining cases are simpler.

(3) Apply the WEAK-rule as many times as needed to the bottom of a deduction of $\vdash_{[]} P \blacktriangleright \Gamma$, to obtain $\vdash_{[]} P \blacktriangleright \Gamma'$.

(4) By part 2 of this lemma $\vdash_{[]} P \blacktriangleright \Gamma \restriction \mathrm{fn}\, P$. Noticing that, since x is not free in P, $\Gamma/x \supseteq \Gamma \restriction \mathrm{fn}\, P$, the result follows from part 3 above. □

REMARK 3.14. Part 2 of the above lemma allows to consider only deductions of P-typings. Furthermore, the proof of the same part shows that, in deductions of P-typings, the only possible use for the WEAK-rule is right before the OBJ-rule, in order to introduce assignments on the names bound by methods but not free in the method's body. This consideration justifies the following convention that will make subsequent proofs easier.

DEDUCTION CONVENTION 3.15. The WEAK-rule is used in deductions solely before the OBJ-rule, and exactly as many times as needed to place typings in the form required by the antecedent of the OBJ-rule.

The second property of system $\mathrm{TA}_{[]}$ allows substituting a name by another name both in a term and in its typing. Recall that, by the definition 3.7 of typing substitution, $\Gamma[z/x]$ is only defined when $z \notin \operatorname{dom}\Gamma$, or when $x, z \in \operatorname{dom}\Gamma$ and $\Gamma z = \Gamma x$.

PROPOSITION 3.16 (Substitution lemma). *If* $\vdash_{[]} P \blacktriangleright \Gamma$, $x \in \operatorname{fn} P$, *and* $\Gamma[z/x]$ *is defined, then* $\vdash_{[]} P[z/x] \blacktriangleright \Gamma[z/x]$.

PROOF. Substitute z for x throughout the given deduction. The substitution creates from a MSG-axiom a new MSG-axiom, and from an instance of the REPL-rule or the WEAK-rule a new instance of the REPL-rule or the WEAK-rule. For the rules containing side conditions on typing compatibility, notice that by proposition 3.8, $\Gamma \asymp \Delta$ implies $\Gamma[z/x] \asymp \Delta[z/x]$. Then the substitution creates from an instance of the COMP-rule an instance of the COMP-rule. Finally, given that $x \in \operatorname{fn} P$, the substitution also creates from an instance of the OBJ-rule or the SCOP-rule an instance of the OBJ-rule or the SCOP-rule. □

The third property ensures that congruent terms have the same typings.

LEMMA 3.17 (Congruence lemma). *If* $\vdash_{[]} P \blacktriangleright \Gamma$ *and* $P \equiv Q$, *then* $\vdash_{[]} Q \blacktriangleright \Gamma$.

PROOF. A case analysis on the rules comprising the congruence relation $\equiv$. The case of α-congruent terms is by induction on the length of deductions; the remaining cases by transformations on deduction trees. The five items below correspond to the five items in the definition 2.10 of $\equiv$.

(1) For the alpha-conversion, the nontrivial cases are when the last rule is associated with a binding operator. For the SCOP-rule, suppose $\vdash_{[]} P \blacktriangleright \Gamma$. By the substitution lemma 3.16, $\vdash_{[]} P[z/x] \blacktriangleright \Gamma[z/x]$, and by the SCOP-rule $\vdash_{[]} P[z/x] \blacktriangleright \Gamma/z$. The case of the OBJ-rule follows by a similar reasoning.

(2) For the commutativity of ",", exchange the two patterns below throughout the given deduction, wherever necessary.

$$
\begin{array}{c}
\dfrac{\begin{array}{cc}\vdots & \vdots\\ P \blacktriangleright \Gamma & Q \blacktriangleright \Delta\end{array}}{P,Q \blacktriangleright \Gamma \cup \Delta}\ (\Gamma \asymp \Delta)\\
\vdots
\end{array}
\qquad\qquad
\begin{array}{c}
\dfrac{\begin{array}{cc}\vdots & \vdots\\ Q \blacktriangleright \Delta & P \blacktriangleright \Gamma\end{array}}{Q,P \blacktriangleright \Delta \cup \Gamma}\ (\Delta \asymp \Gamma)\\
\vdots
\end{array}
$$

The associativity follows by a reasoning similar to the above, by noting that, by proposition 3.6(2), one has ($\Delta \asymp \Gamma$ and $\Delta \cup \Gamma \asymp \Theta$) if and only if ($\Delta \asymp \Theta$ and $\Gamma \asymp \Delta \cup \Theta$).

(3) Similar to the associativity of ",".

(4) Exchange the two patterns below throughout the given deduction, wherever necessary. Since $x \notin \mathrm{fn}\, Q$, by the deduction convention $x \notin \mathrm{dom}\, \Delta$. One then has $(\Gamma \cup \Delta)/x = \Gamma/x \cup \Delta$, and ($\Gamma/x \asymp \Delta$ if and only if $\Gamma \asymp \Delta$).

$$
\begin{array}{c}
\dfrac{\begin{array}{cc}\dfrac{\begin{array}{c}\vdots\\ P \blacktriangleright \Gamma\end{array}}{\nu x P \blacktriangleright \Gamma/x} & \begin{array}{c}\vdots\\ Q \blacktriangleright \Delta\end{array}\end{array}}{\nu x P, Q \blacktriangleright \Gamma/x \cup \Delta}\ (\Gamma/x \asymp \Delta)\\
\vdots
\end{array}
\qquad\qquad
\begin{array}{c}
\dfrac{\dfrac{\begin{array}{cc}\vdots & \vdots\\ P \blacktriangleright \Gamma & Q \blacktriangleright \Delta\end{array}}{P,Q \blacktriangleright \Gamma \cup \Delta}\ (\Gamma \asymp \Delta)}{\nu x(P,Q) \blacktriangleright \Gamma/x \cup \Delta}\\
\vdots
\end{array}
$$

Case $\nu Q \equiv Q$ is easier.

(5) Exchange the two patterns below throughout the given deduction, wherever necessary.

$$
\begin{array}{c}
\vdots\\
\dfrac{P \blacktriangleright \Gamma}{!P \blacktriangleright \Gamma}\\
\hline
\vdots
\end{array}
\qquad\qquad
\begin{array}{c}
\dfrac{\begin{array}{cc}\begin{array}{c}\vdots\\ P \blacktriangleright \Gamma\end{array} & \dfrac{\begin{array}{c}\vdots\\ P \blacktriangleright \Gamma\end{array}}{!P \blacktriangleright \Gamma}\end{array}}{P,\, !P \blacktriangleright \Gamma}\ (\Gamma \asymp \Gamma)\\
\vdots
\end{array}
$$

$\square$

The fourth property says that subterms of typable terms are typable.

PROPOSITION 3.18 (Typability of subterms). *If* $\vdash_{[]} P \blacktriangleright \Gamma$ *and* Q *is a subterm of* P*, then* $\vdash_{[]} Q \blacktriangleright \Delta$ *for some typing* Δ*.*

PROOF. A simple induction on the structure of P, simplified by using the deduction-convention. □

REMARK 3.19. The statement of the above result is in a direct correspondence with typability of subterms for the λ-calculus, as in Barendregt [**6**], proposition 3.1.9. There is a stronger result in $\mathrm{TA}_{[]}$ that says:

If $\vdash_{[]} P \blacktriangleright \Gamma$ and Q is a subterm of P, then $\vdash_{[]} Q \blacktriangleright \Gamma$.

Proposition 3.18 is however all I need to prove the subject-reduction property.

The most important property of system $\mathrm{TA}_{[]}$ ensures that reduction preserves typings.

THEOREM 3.20 (Subject-reduction). *If* $\vdash_{[]} P \blacktriangleright \Gamma$ *and* $P \twoheadrightarrow Q$*, then* $\vdash_{[]} Q \blacktriangleright \Gamma$*.*

PROOF. By induction on reduction $\twoheadrightarrow$, using results 3.13, 3.16, 3.17, and 3.18. When $P \equiv Q$, or when the reduction ends with the STR-rule, the result follows from the congruence lemma 3.17. For the COMM-axiom, suppose $\vdash_{[]} \nu\tilde{x}(a \triangleright M, a \triangleleft C, \tilde{P}) \blacktriangleright \Gamma$. By typability of subterms 3.18, one has $\vdash_{[]} a \triangleright M, a \triangleleft C \blacktriangleright \Delta$, for some typing Δ. I argue that

$$\vdash_{[]} M \bullet C \blacktriangleright \Delta.$$

Then, I replace, in the deduction of $\vdash_{[]} \nu\tilde{x}(a \triangleright M, a \triangleleft C, \tilde{P}) \blacktriangleright \Gamma$, the deduction of $\vdash_{[]} a \triangleright M, a \triangleleft C \blacktriangleright \Delta$ by that of $M \bullet C \blacktriangleright \Delta$, thus obtaining a deduction of $\vdash_{[]} \nu\tilde{x}(M \bullet C, \tilde{P}) \blacktriangleright \Gamma$.

To prove the claim, let $C = l_i : \tilde{v}$ and $M = [l_1 : (\tilde{x}_1)\, P_1 \,\&\, \cdots \,\&\, l_n : (\tilde{x}_n)\, P_n]$, and make $\alpha = [l_1 \colon \tilde{\alpha}_1, \ldots, l_n \colon \tilde{\alpha}_n]$. By the deduction-convention one has the following deduction.

$$\dfrac{\dfrac{\begin{matrix}\vdots & & \vdots \\ P_1 \blacktriangleright \Gamma_1 \cdot \tilde{x}_1 \colon \tilde{\alpha}_1 & \cdots & P_n \blacktriangleright \Gamma_n \cdot \tilde{x}_n \colon \tilde{\alpha}_n\end{matrix}}{a \triangleright M \blacktriangleright \{a \colon \alpha\} \cup \Gamma_1 \cup \cdots \cup \Gamma_n} \qquad a \triangleleft C \blacktriangleright \tilde{v} \colon \tilde{\alpha}_1 \cdot a \colon \alpha}{a \triangleright M, a \triangleleft C \blacktriangleright \{\tilde{v} \colon \tilde{\alpha}_1, a \colon \alpha\} \cup \Gamma_1 \cup \cdots \cup \Gamma_n = \Delta}$$

For each x_{1_i} in $\tilde{x}_1$, and v_i in $\tilde{v}$, $i = 1, \ldots, \mathrm{length}(\tilde{v})$, one has that *either* $x_{1_i} \in \mathrm{fn}\, P_1$, and hence by the substitution lemma 3.16,[1]

$$\vdash_{[]} P_1[v_i/x_{1_i}] \blacktriangleright (\Gamma_1 \cdot \tilde{x}_1 \colon \tilde{\alpha}_1)/x_{1_i} \cdot v_i \colon \alpha_{1_i} \subseteq \Delta,$$

or else $x_{1_i} \notin \mathrm{fn}\, P_1$ and by lemma 3.13(4) $\vdash_{[]} P_1 \blacktriangleright (\Gamma_1 \cdot \tilde{x}_1 \colon \tilde{\alpha}_1)/x_{1_i}$, and since $P_1 = P_1[v_i/x_{1_i}]$, one has

$$\vdash_{[]} P_1[v_i/x_{1_i}] \blacktriangleright (\Gamma_1 \cdot \tilde{x}_1 \colon \tilde{\alpha}_1)/x_{1_i} \subseteq \Delta.$$

Complete the proof by using the WEAK-rule as many as times as needed to obtain $\vdash_{[]} P_1\{\tilde{v}/\tilde{x}_1\} \blacktriangleright \Delta$. $\square$

REMARK 3.21. The converse of subject-reduction does not hold in general. Two cases arise.

(1) Non-typable terms can reduce to typable ones. There are terms P and Q such that

$$P \twoheadrightarrow Q \text{ and } Q \text{ is typable, but } P \text{ is not typable.}$$

Proof. Take $P = a \lhd l : a, a \rhd [l : (x)\,\mathbf{0}]$. Then $P \to \mathbf{0}$ and $\vdash_{[]} \mathbf{0}$, but P is not typable because of the recurring name-structure in a (cf. example 3.10(4)). Nonetheless, the recursive type system to be introduced in chapter 4 allows typing P.

For a counterexample that the recursive type system is not able to fix, take $P = a \rhd [l\colon\mathbf{0}],\ a \lhd l,\ a \lhd m$. Then $P \to (\mathbf{0}, a \lhd m)$, and $\vdash_{[]} \mathbf{0}, a \lhd m \blacktriangleright \{a\colon [m\colon \varepsilon]\}$. But P is not typable since $\{a\colon [l\colon \varepsilon]\}$, the only P-typing for the term $(a \lhd l, a \rhd [l\colon\mathbf{0}])$, is not compatible with $\{a\colon [m\colon \varepsilon, \ldots]\}$, the only P-typing for $a \lhd m$. Notice that P is congruent to $a \rhd [l\colon\mathbf{0}],\ a \lhd m,\ a \lhd l$ and that $[l\colon\mathbf{0}] \bullet m$ is not defined. This is an example of a runtime error.

(2) Free names may be lost in the course of reduction. There are terms P and Q, and a typing Γ such that

$$P \twoheadrightarrow Q \text{ and } \vdash_{[]} Q \blacktriangleright \Gamma, \text{ but } \not\vdash_{[]} P \blacktriangleright \Gamma.$$

[1] Notice that one is in the conditions of the substitution lemma: by the name-convention 2.7, $v_i \notin \{\tilde{x}_1\}$. Further, if $v_i \in \mathrm{dom}\,\Gamma_1$, one has, by the side condition of the COMP-rule in the above deduction, $\tilde{v} \colon \tilde{\alpha}_1 \asymp \Gamma_1$, implying that $\alpha_i = \Gamma v_i$.

Proof. Take $P = a \triangleright [l : \mathbf{0}],\ a \triangleleft l$. Then $P \to \mathbf{0}$, and $\vdash_{[]} \mathbf{0}$, but, by lemma 3.13(1), the empty typing is not a well-typing for P.

A consequence of the subject-reduction property is that a typable programs do not run into errors during execution. Stemming from a single source (message application not being defined for a given message and object) two kinds of runtime errors can be distinguished:

1. the label in a message labels no method in the target object, and
2. the length of the sequence of names in a message does not match that of the formal parameters of the corresponding method.

DEFINITION 3.22 (Runtime errors). A term P contains a possible *runtime error*, notation $P \in \mathrm{Err}$, if there exists a term Q such that $P \twoheadrightarrow Q = \nu\tilde{x}(a \triangleright M, a \triangleleft C, \tilde{P})$ and $M \bullet C$ is not defined; that is, if $C = l_i : \tilde{v}$, $M = [l_1 : (\tilde{x}_1)\, P_1 \& \cdots \& l_n : (\tilde{x}_n)\, P_n]$, and either $i > n$, or $1 \leq i \leq n$ but $\mathrm{length}\, \tilde{v} \neq \mathrm{length}\, \tilde{x}_i$.

COROLLARY 3.23. *If P is typable, then $P \notin \mathrm{Err}$.*

PROOF. By subject-reduction, $\nu\tilde{x}(a \triangleright M, a \triangleleft C, \tilde{P})$ is typable. Suppose that $C = l : \tilde{v}$ and $M = [l_1 : (\tilde{x}_1)\, P_1 \,\&\, \cdots \,\&\, l_n : (\tilde{x}_n)\, P_n]$. By typability of subterms and the MSG-axiom, one has $\vdash_{[]} a \triangleleft C \ \blacktriangleright\ \tilde{v} \colon \tilde{\alpha} \cdot a \colon \beta$, for some type β of the form $[l \colon \tilde{\alpha}, \ldots]$. By typability of subterms and the OBJ-rule one also has $\vdash_{[]} a \triangleright M \ \blacktriangleright\ \Gamma$, for some typing Γ such that $\Gamma a = [l_1 \colon \tilde{\alpha}_1, \ldots, l_n \colon \tilde{\alpha}_n]$. By the side-condition of the COMP-rule, $\beta = [l_1 \colon \tilde{\alpha}_1, \ldots, l_n \colon \tilde{\alpha}_n]$. Hence l is one of the $l_1, \ldots, l_n$, say l_1, and thus $\tilde{\alpha}_1 = \tilde{\alpha}$ implying that $\mathrm{length}\, \tilde{v} = \mathrm{length}\, \tilde{x}_1$. ☐

Decidability of typing assignment and computability of typing inference are the last properties considered in this chapter. Their proofs, however, must be postponed until the end of section 6, for some additional machinery is needed.

THEOREM 3.24 (Decidability of typing assignment in $\mathrm{TA}_{[]}$). *Given a term P and a typing Γ, it can be decided whether $\vdash_{[]} P \ \blacktriangleright\ \Gamma$.*

THEOREM 3.25 (Computability of typing inference in $\mathrm{TA}_{[]}$). *Given a term P, it can be decided whether there is a typing Γ such that $\vdash_{[]} P \ \blacktriangleright\ \Gamma$. Moreover, one such Γ can be computed from P.*

5. Notes

The idea of introducing a discipline in the roles names play in computations is due to Milner [**38, 39**], in the context of the π-calculus. When one goes from the monadic into the polyadic π-calculus, that is when one allows a name to carry not a single name but an arbitrary long sequence of names, potential communication errors arise in the form of mismatch between the length of the names sent and of those expected.

In order to characterize well-behaving programs, Milner introduced the notion of *sorts* for names. A *sorting function*, from sorts to sequences of sorts, describes the sequence of names a name of a given sort may carry. A sort-check system allows to verify whether a process is well-sorted against a given sorting. Sorts and types are indeed close notions. There is however an important difference: sorts induce a notion of *name-matching*, as opposed to the notion of *structure-matching* for types [**64, 65**].

A typing assignment system for the polyadic π-calculus, based on type assignment systems for the λ-calculus, was presented by Honda and myself [**64, 65**]. While both the sort-check and the typing-assignment systems ensure that well-sorted (or typable) programs do not encounter errors at runtime, the basic philosophy of the two systems is different. The sort-check system *checks* whether a process is "OK" against a given sorting, in particular requiring bound names to be distinct from the free names. The typing inference system *builds* a typing for a process. The typing lemma 3.13 does not hold in the sort-check system. See reference [**64, 65**] for further details.

A more refined sorting system that distinguishes names used solely for input, from those used solely for output, and those used for both input and output, thus inducing an interesting notion of subtyping, was presented by Pierce and Sangiorgi [**46**].

The simple typing assignment system for the calculus of objects was first proposed by Tokoro and myself [**66**]. Most of the properties of this system are in a simple correspondence with those of the type system for the polyadic π-calculus mentioned above. The first proof of the subject-reduction property for name-passing calculi appeared in reference [**64**] and was first published in reference [**65**].

Proofs for most of the properties discussed in this chapter appeared in reference [**64**] (in the context of the π-calculus), but were mostly conducted by induction on the length of derivations. Proofs by transformation on deduction trees are much more intuitive; the ones in this thesis were inspired on Hindley and Seldin [**26**]. Finally, I would like to note that the proof of the subject-reduction property for the polyadic π-calculus [**65**] implicitly assumes the name-convention.

An important property that does not carry from the type system for the polyadic π-calculus into the simple type system for the calculus of objects is the existence and computability of principal typings. This will be the theme of chapter 5.

CHAPTER 4

Recursive Typing Assignment

This chapter presents an extension to the simple type system $\mathrm{TA}_{\square}$ that allows typing terms with a recurring name structure. The recursive type system $\mathrm{TA}_{\square\mu}$ extends the class of typable terms, while preserving all the syntactical properties of system $\mathrm{TA}_{\square}$ studied in chapter 3.

1. Recursive Types

Although sometimes meaning and useful, terms containing recurring name-structures cannot be typed in system $\mathrm{TA}_{\square}$ (cf. remark 3.12). Recalling example 3.10(4), if we had a type α such that $\alpha = [\mathit{val}\colon [\mathit{cons}\colon t\alpha]]$, then we could assign α to name c in term $c \rhd [\mathit{val}\colon (x)\, x \lhd \mathit{cons}\colon vc]$. Solutions of the above equation are axiomatically written $\mu u.[\mathit{val}\colon [\mathit{cons}\colon tu]]$, and interpreted as the infinite tree that solves the equation $\alpha = [\mathit{val}\colon [\mathit{cons}\colon t\alpha]]$. Intuitively, a type $\mu t.\alpha$ can be seen as the infinite unfolding obtained by replacing infinitely many times t by α in α.

DEFINITION 4.1 (Recursive types). The set $\mathbf{T}_{\square\mu}$ of *recursive types* is defined by adding to the syntax of simple types in definition 3.1, a production $\mu t.\alpha$, for any type-variable t in $\mathbf{V}$ and any type α in $\mathbf{T}_{\square\mu}$. In other words, the set $\mathbf{T}_{\square\mu}$ of recursive types is inductively defined by the following grammar.

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

Lower case Greek letters $\alpha, \beta, \ldots$ will also be used to denote recursive types.

2. Recursive Types as Regular Infinite Trees

DEFINITION 4.2 (Infinite trees).

1. A *tree* over a ranked alphabet F is a partial mapping, $\rho : \mathcal{N}_+^* \to F$, from the set of strings over positive integers $\mathcal{N}_+$ into symbols in F, satisfying a tree domain condition:
   (a) The domain of ρ is prefix closed (that is, if $\tilde{\imath}, \tilde{\jmath} \in \mathcal{N}_+^*$, and $\tilde{\imath}\tilde{\jmath} \in \operatorname{dom}\rho$, then $\tilde{\imath} \in \operatorname{dom}\rho$) and non empty,
   (b) All the siblings at the left of a node in ρ are nodes in ρ (that is, if $\tilde{\imath} \in \mathcal{N}_+^*$, $j, k \in \mathcal{N}_+$, $j \leq k$, and $\tilde{\imath}k \in \operatorname{dom}\rho$, then $\tilde{\imath}j \in \operatorname{dom}\rho$), and
   (c) a node of rank k has exactly k subtrees, numbered 1 to k (that is, if $\rho(\tilde{\imath}) = f$ of rank k, then for $j \in \mathcal{N}_+$, $\tilde{\imath}j \in \operatorname{dom}\rho$ if and only if $1 \leq j \leq k$).
2. If $\rho_1, \ldots, \rho_n$ are n trees ($n \geq 1$) and the rank of the symbol $f \in F$ is n, write $f(\rho_1, \ldots, \rho_n)$ for the tree ρ such that

$$\begin{array}{rcl} \rho(\varepsilon) & = & f, \\ \rho(i\tilde{\jmath}) & = & \rho_i(\tilde{\jmath}) \quad \text{if } 1 \leq i \leq n \text{ and } \tilde{\jmath} \in \mathcal{N}_+^*, \\ \rho(\tilde{\jmath}) & \multicolumn{2}{l}{\text{is undefined otherwise.}} \end{array}$$

3. The *subtree of a tree ρ issued from node $\tilde{\imath}$* is the tree $\lambda\tilde{\jmath}.\rho(\tilde{\imath}\tilde{\jmath})$. The set of all *subtrees* of a tree ρ is $\{\rho \text{ issued from } \tilde{\imath} \mid \tilde{\imath} \in \operatorname{dom}\rho\}$. A tree is *regular* is the set of all its subtrees is finite.
4. A *substitution on infinite trees* is a mapping from variables into trees over F, the set of variables being considered a subset of F. Given a substitution s and a tree ρ, the tree $s\rho$ is the result of the simultaneous substitution of tree st for each occurrence of variable t in ρ. Write $[\zeta/t]\rho$ for the tree $s\rho$, when substitution s assigns the tree ζ to variable t and leaves unchanged all other variables. It is easy to see that if both ρ and ζ are regular trees, then so is $[\zeta/t]\rho$.

DEFINITION 4.3. 1. Assume that the set $\mathbf{L}$ of labels is equipped with a *total strict order* $<$, and with a *ranking function* rank : $\mathbf{L} \to \mathcal{N}$. The set $\mathbf{F}$ of *tree labels* is the set of strings of labels in $\mathbf{L}$, such that $l_1 \cdots l_n \in \mathbf{F}$ only if $l_1 < \cdots < l_n$. As usual, the empty string is ε. Observe that, since by definition $\mathbf{L}$ is disjoint from $\mathbf{V}$, also $\mathbf{F}$ is disjoint from $\mathbf{V}$.

2. Let $\mathrm{rank}(\varepsilon) = 0$, and $\mathrm{rank}(l_1 \cdots l_n) = \sum_{i=1}^{n} \mathrm{rank}(l_i)$, for $n \geq 1$. Type-variables in $\mathbf{V}$ are symbols of rank 0.
3. Denote by $\mathbf{T}_{\mathrm{R}}$ the set of all regular trees over $\mathbf{F} \cup \mathbf{V}$.

Recursive types in $\mathbf{T}_{[]\mu}$ are interpreted as regular infinite trees in $\mathbf{T}_{\mathrm{R}}$.

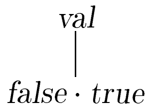
DEFINITION 4.4. The translation $(-)^* : \mathbf{T}_{[]\mu} \to \mathbf{T}_{\mathrm{R}}$, from recursive types into regular infinite trees, is the partial mapping inductively defined by

$$\begin{aligned} t^* &= t, \\ []^* &= \varepsilon, \\ [l_1\colon \tilde{\alpha}_1, \ldots, l_n\colon \tilde{\alpha}_n]^* &= l_1 \cdots l_n(\tilde{\alpha}_1^*, \ldots, \tilde{\alpha}_n^*) \qquad \text{for } l_1 < \cdots < l_n \text{ and } n \geq 1, \\ \mu t.\alpha^* &= \text{the unique tree } \rho \text{ such that } \rho = [\rho/t]\alpha^* \text{ and } \alpha^* \neq t, \end{aligned}$$

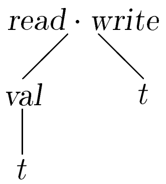
where $\tilde{\alpha}^*$ represents the sequence of trees $\alpha_1^*, \ldots, \alpha_k^*$ if $\tilde{\alpha} = \alpha_1 \cdots \alpha_k$. Say ρ *defines* α when $\alpha^* = \rho$.

The tree associated with a recursive type is undefined when, in the last case, $\alpha^* = t$, or when, in the third and the last case, at least one of $\alpha^*, \tilde{\alpha}_1^*, \ldots, \tilde{\alpha}_n^*$ is undefined.

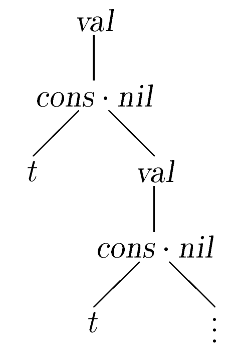
EXAMPLES 4.5. (1) Let *false* < *true*. Then the below tree defines the type [*val*: [*true*, *false*]] of boolean objects.



(2) If *read* < *write*, the tree defining the type [*read*: [*val*: t], *write*: t] of the buffer cell is depicted below.



(3) If *cons* < *nil*, below is the tree defining the list type $\mu u.$[*val*: [*cons*: tu, *nil*]], and also the type [*val*: $\mu u.$[*cons*: t[*val*: u], *nil*]].



Observe that types $\mu u.[val\colon [cons\colon tu, nil]]$ and $[val\colon \mu u.[cons\colon t[val\colon u], nil]]$ cannot be obtained from each other by just unfolding their definitions.

The following proposition shows that the translation from types into regular infinite trees is well defined.

PROPOSITION 4.6. *Trees defining recursive types are regular.*

PROOF. By induction on the structure of types. The base cases are trivial; the case of records follows directly by induction. For types of the form $\mu t.\alpha$, if α^* is a type-variable (different from t), the tree ρ such that $\rho = [\rho/t]\alpha^* = \alpha^*$ is clearly a regular tree. Otherwise, by induction, α^* is regular, and thus the system of equations $\langle t \doteq \alpha^* \rangle$ is a generalized regular Greibach system. As such, it has a unique solution in $\mathbf{T}_{\mathrm{R}}$ whose only component ρ is regular (Courcelle [**18**], theorem 4.3.1). $\square$

REMARK 4.7. The mapping $(-)^*$ is surjective but not one-to-one. In general, for every regular infinite tree ρ in $\mathbf{T}_{\mathrm{R}}$ there are many types α in $\mathbf{T}_{[]\mu}$ such that $\alpha^* = \rho$. For example, as seen in example 4.5(3), both the type $\mu u.[val\colon [cons\colon tu, nil]]$ and the type $[val\colon \mu u.[cons\colon t[val\colon u], nil]]$ are translated into the same tree. Notice also that different permutations of the same record type are translated into the same tree.

The interpretation of recursive types as regular infinite trees yields a natural notion of equivalence on recursive types.

DEFINITION 4.8 (Type equivalence). Two types α and β in $\mathbf{T}_{[]\mu}$ are *equivalent*, notation $\alpha \approx \beta$, if they are defined by the same tree, that is, if $\alpha^* = \beta^*$.

LEMMA 4.9. *Type equivalence is decidable.*

PROOF. Courcelle shows that every regular tree is a component of the unique solution of some system of regular equations ([**18**], theorem 4.2.1). Furthermore, his proof makes it clear that the system is unique up to renaming of the unknowns in the equations. Courcelle, Kahn, and Vuillemin show that the equivalence of systems of regular equations is decidable ([**17**], theorem 2). □

Recursive types are interpreted as regular infinite trees. There are however types whose associated tree is undefined. The tree $(\mu t.\alpha)^*$ is undefined when $\alpha^* = t$. In particular, $(\mu t.t)^*$ is undefined. More generally, the tree associated with type $\mu t_1.\cdots.\mu t_n.t_i$, for $n \geq 1$, is undefined when $i \leq n$.

There are also recursive types equivalent to non-recursive types. In particular, a type $\mu t.t'$ is equivalent to t' if $t' \neq t$. More generally, a type $\mu t_1.\cdots.\mu t_n.t_i$, for $n \geq 1$, is equivalent to t_i when $i > n$.

PROPOSITION 4.10. *Let n be a positive integer. Type $\mu t_1.\cdots.\mu t_n.t_i$ is undefined if $i \leq n$, and equivalent to t_i otherwise.*

PROOF. By induction on n. When $n = 1$, by definition of $(-)^*$, one has that $\mu t_1.t_i^*$ is undefined when $i = 1$, and is the tree ρ such that $\rho = [\rho/t_1]t_i = t_i$ otherwise. When $n > 1$, one has that $(\mu t_1.\cdots.\mu t_n.t_i)^*$ is undefined when $(\mu t_2.\cdots.\mu t_n.t_i)^*$ is undefined, and is the tree ρ such that $\rho = [\rho/t_n](\mu t_2.\cdots.\mu t_n.t_i)^*$ otherwise. In the latter case, by induction, $(\mu t_2.\cdots.\mu t_n.t_i)^* = t_i$, and hence $\rho = t_i$. □

Every regular tree in $\mathbf{T}_{\mathrm{R}}$ defines a type in $\mathbf{T}_{[]\mu}$. This fact allows to prove facts about recursive types using regular trees, and will be particularly useful in the next chapter to establish the decidability of typing assignment and computability of typing inference in the recursive setting.

LEMMA 4.11 (Braquelaire and Courcelle [**12**], corollary 5.6). *For every regular tree ρ in $\mathbf{T}_R$ one can construct a recursive type in $\mathbf{T}_{[]\mu}$ defining ρ.*

There is now a *binding on type-variables.* An occurrence of a type-variable in a type is *bound* if it occurs in the part α of $\mu t.\alpha$; otherwise it is called *free.* Clearly, a type-variable may occur both free and bound in a type: take for example t in type $[l\colon t\ \mu t.[l\colon t]]$. To simplify the treatment of substitution I follow the convention that bound type-variables in types appearing in mathematical contexts are chosen different from the free type-variables (cf. name convention 2.7, and variable convention in Barendregt [**5**]).

The substitution of a type for a type-variable in a given type in $\mathbf{T}_{[]\mu}$ may now be defined without fear of capture of free type-variables under bindings. The exact definition is by induction on the structure of types, in much the same way as in the definition 2.8 of name substitution.

PROPOSITION 4.12. *If $\alpha^* = \rho$ and s is a substitution, then $(s\alpha)^* = s\rho$.*

PROOF. By induction on the structure of types, using the convention on type-variables. The case of type-variables is trivial. For records, suppose $\tilde{\alpha}_i^* = \tilde{\rho}_i$, for $1 \leq i \leq n$. Then,

$$\begin{aligned}
(s[l_1\colon \tilde{\alpha}_1, \ldots, l_n\colon \tilde{\alpha}_n])^* &= [l_1\colon s\tilde{\alpha}_1, \ldots, l_n\colon s\tilde{\alpha}_n]^* \\
&= l_1 \cdots l_n((s\tilde{\alpha}_1)^*, \ldots, (s\tilde{\alpha}_n)^*) \\
&= l_1 \cdots l_n(s\tilde{\rho}_1, \ldots, s\tilde{\rho}_n) \\
&= s(l_1 \cdots l_n(\tilde{\rho}_1, \ldots, \tilde{\rho}_n)).
\end{aligned}$$

For recursive types, suppose $\alpha^* = \rho$. Then $(s\mu t.\alpha)^* = (\mu t.s\alpha)^* = \rho'$ for some tree ρ' such that $\rho' = [\rho'/t]((s\alpha)^*) = [\rho'/t](s\rho)$. Since $st = t$, one has $\rho' = s([\rho'/t]\rho) = s((\mu t.\alpha)^*)$. □

Type equivalence is preserved by substitution.

PROPOSITION 4.13. *If $\alpha \approx \beta$ and s is a substitution, then $s\alpha \approx s\beta$.*

PROOF. A simple induction on the structure of types, using proposition 4.12. □

3. Recursive Typing Assignment System $\mathbf{TA}_{[]\mu}$

DEFINITION 4.14 (Recursive typing assignment system $\mathrm{TA}_{[]\mu}$). The recursive typing assignment system $\mathrm{TA}_{[]\mu}$ is composed of the following axiom

(MSG_μ)

$$a \lhd l_1 : \tilde{v} \blacktriangleright \{a\colon \beta, \tilde{v}\colon \tilde{\alpha}_1\} \qquad \text{(condition: } \beta \approx [l_1\colon \tilde{\alpha}_1, \ldots, l_n\colon \tilde{\alpha}_n]),$$

and the rules in the definition 3.9 of the simple typing assignment system $\mathrm{TA}_{[]}$, together with the following rule.

$$(\approx) \qquad \frac{P \blacktriangleright \Gamma \cdot x\colon \alpha}{P \blacktriangleright \Gamma \cdot x\colon \beta} \qquad \text{(condition: } \alpha \approx \beta).$$

Write $\vdash_{[]\mu} P \blacktriangleright \Gamma$ if there is a deduction of $P \blacktriangleright \Gamma$ in $\mathrm{TA}_{[]\mu}$.

REMARK 4.15. It is also possible to define a recursive typing assignment system by simply adding the $\approx$-rule to system $\mathrm{TA}_{[]}$. The MSG_μ-axiom, however, allows to type messages whose target is contained in the message's communication. For example, one has $\vdash_{[]\mu} a \lhd l : a \blacktriangleright \{a\colon \mu t.[l\colon t]\}$ since $\mu t.[l\colon t] \approx [l\colon \mu t.[l\colon t]]$.

EXAMPLE 4.16. I showed in example 3.10(4) that term $\mathrm{Cons}\, cvc$ is not typable in system $\mathrm{TA}_{[]}$ because of its recurring name structure. The recursive type assignment system provides for typing such terms. Recall that $\mathrm{Cons}\, cvc$ is, by definition, the term $c \rhd [\mathit{val}\colon (x)\, x \lhd \mathit{cons}\colon vc]$.

Let $\alpha = \mu u.[\mathit{val}\colon [\mathit{cons}\colon tu]]$; notice that $\alpha \approx [\mathit{val}\colon [\mathit{cons}\colon t\alpha]]$. Starting with the MSG-axiom, and applying the $\approx$, the OBJ, and the $\approx$-rule in succession, one has the following deduction.

$$\cfrac{\cfrac{\cfrac{x \lhd \mathit{cons}\colon vc \blacktriangleright \{x\colon [\mathit{cons}\colon t\alpha], v\colon t, c\colon \alpha\}}{x \lhd \mathit{cons}\colon vc \blacktriangleright \{x\colon [\mathit{cons}\colon t\alpha], v\colon t, c\colon [\mathit{val}\colon [\mathit{cons}\colon t\alpha]]\}}}{c \rhd [\mathit{val}\colon (x)\, x \lhd \mathit{cons}\colon vc] \blacktriangleright \{v\colon t, c\colon [\mathit{val}\colon [\mathit{cons}\colon t\alpha]]\}}}{c \rhd [\mathit{val}\colon (x)\, x \lhd \mathit{cons}\colon vc] \blacktriangleright \{v\colon t, c\colon \alpha\}}$$

Also, noticing that $[\mathit{cons}\colon t\alpha] \approx [\mathit{cons}\colon t[\mathit{val}\colon [\mathit{cons}\colon t\alpha]]]$, one has the following instance of the MSG_μ-axiom

$$x \lhd \mathit{cons}\colon vc \blacktriangleright \{x\colon [\mathit{cons}\colon t\alpha], v\colon t, c\colon [\mathit{val}\colon [\mathit{cons}\colon t\alpha]]\}$$

thus saving the first application of the $\approx$-rule in the above deduction. A simple adaptation of the above deduction shows that

$$\vdash_{[]\mu} \mathrm{Cons}\, cvc \blacktriangleright \{v\colon t, c\colon \mu u.[val\colon [cons\colon tu, nil]]\}.$$

Let $\beta = [val\colon \mu u.[cons\colon t[val\colon u]]]$. Noticing that $\mu u.[cons\colon t[val\colon u]] \approx [cons\colon t\beta]$, one has the following deduction, starting with the Msg_μ-axiom and applying the Obj-rule.

$$\frac{x \lhd cons{:}vc \blacktriangleright \{x\colon \mu u.[cons\colon t[val\colon u]], v\colon t, c\colon \beta\}}{c \rhd [val{:}(x)\, x \lhd cons{:}vc] \blacktriangleright \{v\colon t, c\colon \beta\}}$$

Summarizing, one has $\vdash_{[]\mu} \mathrm{Cons}\, cvc \blacktriangleright \{v\colon t, c\colon \alpha\}$ and also $\vdash_{[]\mu} \mathrm{Cons}\, cvc \blacktriangleright \{v\colon t, c\colon \beta\}$. As seen in example 4.5(3), α and β cannot be obtained from each other by unfolding, but they represent the same infinite tree.

All properties of the simple typing assignment system $\mathrm{TA}_{[]}$ discussed in section 4 hold when formulated in the recursive typing assignment system $\mathrm{TA}_{[]\mu}$. Decidability of typing assignment and computability of typing inference, the counterparts of theorems 3.24 and 3.25, also hold in system $\mathrm{TA}_{[]\mu}$; their proofs are postponed until the end of section 6.

Theorem 4.17 (Decidability of typing assignment in $\mathrm{TA}_{[]\mu}$). *Given a term P and a typing Γ, it can be decided whether $\vdash_{[]\mu} P \blacktriangleright \Gamma$.*

Theorem 4.18 (Computability of typing inference in $\mathrm{TA}_{[]\mu}$). *Given a term P, it can be decided whether there is a Γ such that $\vdash_{[]\mu} P \blacktriangleright \Gamma$. Moreover, one such Γ can be computed from P.*

Theorem 4.19. *All the remaining properties of system $TA_{[]}$ discussed in section 4 hold when formulated in system $TA_{[]\mu}$.*

Proof. For the typing lemma, notice that, when the substitution on typings (definition 3.7) is defined, if $P \blacktriangleright \Gamma$ and $P \blacktriangleright \Delta$ are instances for the Msg-axiom and the Msg_μ-axiom respectively, then $\mathrm{dom}\,\Gamma = \mathrm{dom}\,\Delta$. For the $\approx$-rule, notice that the typing of the antecedent and that of the consequent of the rule have the same domain. For the substitution lemma, notice that substituting z for x in an instance of the Msg_μ-axiom or the $\approx$-rule creates an instance of the Msg_μ-axiom or of the $\approx$-rule. Being by induction on the structures of terms, the proof of typability of subterms is not affected by the new rules. The proof that structural congruent processes have the same typings easily accommodates the MSG_μ-axiom and the $\approx$-rule. The proof of subject-reduction is not affected by the new rules, and similarly for the corollary on the lack of runtime errors. $\square$

The main property of the λ-calculus that does not carry from the simple to the recursive type system is strong normalization of typable terms, but there seems to be no counterpart of such result in name-passing calculi.

There is a recursive type that can be assigned to every λ-term, namely $\mu t.t \to t$. It is easy to check that every λ-term M has type $\mu t.t \to t$ under a basis assigning $\mu t.t \to t$ to every variable in M [**7, 16, 15**]. There is no counterpart of such property in the calculus of objects, the reason being that there is not a unique type constructor in $\mathbf{T}_{[]}$, but else a collection of records with a variable number of components. Nonetheless, by fixing a label, say *val*, and by restricting messages and objects to contain exactly this label, one has the following result.

PROPOSITION 4.20. *Let P be a term in $\mathbf{P}$ such that all objects and messages in P are of the form $a \triangleright [\mathrm{val}\colon (x)\, Q]$ and $a \triangleleft \mathrm{val}\colon v$, for arbitrary names a, x, v, and term Q in the conditions of P. Then*

$$\vdash_{[]\mu} P \blacktriangleright \{a\colon \mu t.[\mathit{val}\colon t] \mid a \in \mathrm{fn}\, P\}.$$

PROOF. A simple induction on the structure of P, by noticing that $\mu t.[\mathit{val}\colon t] \approx [\mathit{val}\colon \mu t.[\mathit{val}\colon t]]$ (when P is an object or a message), and that $\Gamma \asymp \Delta$ for any typings Γ and Δ constructed with a single type (when P is an object or a concurrent composition). $\square$

REMARK 4.21. The calculus of objects with objects and messages restricted to contain exactly one label is indeed Honda's ν-calculus [**28, 30**], where one writes $a(x)\,.P$ instead of $a \triangleright [\mathrm{val}\colon (x)\, P]$, and $\leftarrow av$ instead of $a \triangleleft \mathrm{val}\colon v$. A simple adaptation of the above proof yields that all names in ν-terms have the type $\mu t.(t)$, corresponding to the type $\mu t.[\mathit{val}\colon t]$ (see section 1).

Not surprisingly, the same result holds for the monadic π-calculus (see references [**64, 65**] for a type system for the π-calculus). Milner noticed that all names in the monadic π-calculus have the sort NAME under the sorting $\{\text{NAME} \mapsto (\text{NAME})\}$ ([**39**], page 34). This is in harmony with our result since the regular infinite tree $\mu t.(t)^*$ is the only solution of the equation $\text{NAME} = (\text{NAME})$.

REMARK 4.22. A different approach to recursive types was proposed by MacQueen, Plotkin, and Sethi [**35**], by introducing a pair of rules allowing to "fold" and "unfold" a recursive type. An adaptation of their rules to the calculus of objects yields the below rules.

$$(\textsc{Fold}) \qquad \frac{P \blacktriangleright \Gamma \cdot x \colon [\mu t.\alpha/t]\alpha}{P \blacktriangleright \Gamma \cdot x \colon \mu t.\alpha}$$

$$(\textsc{Unfold}) \qquad \frac{P \blacktriangleright \Gamma \cdot x \colon \mu t.\alpha}{P \blacktriangleright \Gamma \cdot x \colon [\mu t.\alpha/t]\alpha}$$

An advantage of using the FOLD and UNFOLD rules is that no interpretation of recursive types is needed. The main drawback, as noted by the authors themselves, is that there are types with the same infinite unfolding that cannot be obtained from one another by using the FOLD and the UNFOLD rules (cf. example 4.5(3)).

This observation leads to the conclusion that a type system incorporating the FOLD and UNFOLD rules, instead of the $\approx$-rule, misses the property of principal types. For example, term Cons cvc can be assigned both the type $\mu u.[\mathit{val}\colon [\mathit{cons}\colon tu]]$ and the type $[\mathit{val}\colon \mu u[\mathit{cons}\colon t[\mathit{val}\colon u]]]$, but neither can be obtained from the other by using the FOLD and UNFOLD rules (cf. example 4.16).

4. Notes

The recursive typing assignment system $\mathrm{TA}_{[]\mu}$ was profoundly influenced by the work of Cardone and Coppo on recursive types for the λ-calculus [**15, 16**], in particular the interpretation of recursive types as regular infinite trees, and the preference of the $\approx$-rule over the FOLD and UNFOLD rules of MacQueen, Plotkin, and Sethi [**35**].

Honda and myself first imported the ideas of Cardone and Coppo into the process-calculi world, namely into the polyadic π-calculus [**64, 65**].

The first publication of the calculus of objects merely points out the possibility of extending the simple type assignment system into the recursive setting [**66**]. The formalization of the recursive typing assignment system appeared later [**59, 62**], where I choose to encode recursive record-types into traditional regular infinite trees (thus missing in some sense the record-like structure of types) in order to be able to use the rich body of results on infinite trees developed mainly by Courcelle and its colleagues [**12, 17, 18**].

The main difference between the above cited works dealing with recursive types (Cardone and Coppo for the λ-calculus [**15, 16**], Honda and myself for the polyadic π-calculus [**64, 65**], and the original presentation of the recursive type system $\mathrm{TA}_{[]\mu}$ [**59, 62**]) and the proposal of this thesis lies in the translation of recursive types into infinite trees. There, a recursive type is seen as the least upper bound of a denumerable increasing sequence of finite trees, starting with the "bottom" tree $\perp$; here the same type is seen the solution of a regular equation, guaranteed to exist by Courcelle's results [**18**]. While the work of Cardone and Coppo is motivated by the interpretation of recursive types in topological models, a simpler formulation is possible if no such interpretation is needed. The interpretation of a recursive type $\mu t.\alpha$ as the only tree that solves the equation $t = \alpha$, taken from Braquelaire and Courcelle [**12**], is much simpler and spares topological considerations.

CHAPTER 5

Kinded Typing Assignment

This chapter introduces type systems where type-variables are assigned *kinds* constraining the types they may be substituted for. The main properties of the kinded type systems are the existence and computability of principal typings, and the correctness with respect to the non-kinded (simple and recursive) type systems, in the sense that the class of terms typable in both systems coincide.

1. Kinds and Kind Assignment

DEFINITION 5.1 (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 $n \geq 0$. Lower case Roman letter k will be used for kinds.

Similarly to methods in terms and to record types, kinds are unordered, so that $\langle l \colon \tilde{\alpha}, l' \colon \tilde{\beta} \rangle$ and $\langle l' \colon \tilde{\beta}, l \colon \tilde{\alpha} \rangle$ represent the same kind.

Intuitively, kinds describe sets of record-types: a kind of the form $\langle l_1 \colon \tilde{\alpha}_1, \ldots, l_n \colon \tilde{\alpha}_n \rangle$ denotes the subset of all 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 5.2 (Kindings).

1. *Kind assignment* is any expression

$$t \colon k,$$

for all type-variables t and kinds k, where t is called the *subject* and k the *predicate* of the assignment.

2. *Kindings* are finite sets of kind assignments where no two assignments have the same type-variable as subject. Upper case Roman letter K will denote a kinding.
3. 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 $n \geq 1$.
4. Pairs of the form (K, Γ) are called *kinded typings.*

DEFINITION 5.3 (Kind assignment to types). A *type* α *has a kind* k *under a kinding* K, notation $K \vdash_{\langle\rangle} \alpha \blacktriangleright k$, if $K \vdash \alpha \blacktriangleright k$ is an instance of one of the two following axioms.

(KRCD)

$$K \vdash [l_1 \colon \tilde{\alpha}_1, \ldots, l_n \colon \tilde{\alpha}_n, \ldots] \blacktriangleright \langle l_1 \colon \tilde{\alpha}_1, \ldots, l_n \colon \tilde{\alpha}_n \rangle$$

(KVAR)

$$K \cdot t \colon \langle l_1 \colon \tilde{\alpha}_1, \ldots, l_n \colon \tilde{\alpha}_n, \ldots \rangle \vdash t \blacktriangleright \langle l_1 \colon \tilde{\alpha}_1, \ldots, l_n \colon \tilde{\alpha}_n \rangle$$

2. Kinded Typing Assignment System $\mathbf{TA}_{[]\langle\rangle}$

DEFINITION 5.4 (Kinded typing assignment system $\mathrm{TA}_{[]\langle\rangle}$). The kinded typing assignment system $\mathrm{TA}_{[]\langle\rangle}$ is a formal theory whose *formulas* are expressions $K \vdash P \blacktriangleright \Gamma$ for all *acyclic* kindings K, terms P, and typings Γ. Its *axioms* are those in the definition 5.3 of kind assignment to types, and the particular case of the OBJ_k-rule when $n = 0$. Its *rules* are those in the definition 3.9 of the simple typing assignment system $\mathrm{TA}_{[]}$ with formulas $P \blacktriangleright \Gamma$ replaced by $K \vdash P \blacktriangleright \Gamma$, together with the rule below.

(MSG$_k$)

$$\frac{K \vdash \beta \blacktriangleright \langle l \colon \tilde{\alpha} \rangle}{K \vdash a \triangleleft l \colon \tilde{v} \blacktriangleright \{a \colon \beta, \tilde{v} \colon \tilde{\alpha}\}}$$

Write $K \vdash_{[]\langle\rangle} P \blacktriangleright \Gamma$ if there is a deduction of $K \vdash P \blacktriangleright \Gamma$. In this case say P is *typable*, call Γ a *well-typing for* P *under kinding* K, and call the pair (K, Γ) a *well kinded-typing* for P. When there is a deduction of $\emptyset \vdash P \blacktriangleright \Gamma$, I shall simply write $\vdash_{[]\langle\rangle} P \blacktriangleright \Gamma$.

EXAMPLES 5.5. (1) One has the following deduction starting with the KRCD-axiom, and using the MSG_k and the OBJ_k-rules.

$$\dfrac{\dfrac{[put\colon t] \blacktriangleright \langle put\colon t\rangle}{b\lhd put\colon x \blacktriangleright \{x\colon t, b\colon [put\colon t]\}}}{a\rhd[put\colon(x)\,b\lhd put\colon x] \blacktriangleright \{a\colon [put\colon t], b\colon [put\colon t]\}}$$

Let $K = \{t'\colon \langle put\colon t, get\colon t''\rangle\}$ be a kinding. One has the following deduction starting with the KVAR-axiom, followed by the MSG_k and the OBJ_k-rule.

$$\dfrac{\dfrac{K \vdash t' \blacktriangleright \langle put\colon t\rangle}{K \vdash b\lhd put\colon x \blacktriangleright \{x\colon t, b\colon t'\}}}{K \vdash a\rhd[put\colon(x)\,b\lhd put\colon x] \blacktriangleright \{a\colon [put\colon t], b\colon t'\}}$$

(2) One has

$$\{t'\colon \langle val\colon t\rangle\} \vdash_{[]\langle\rangle} \mathrm{Cell}\, sv \blacktriangleright \{s\colon [read\colon t', write\colon t], v\colon t\}.$$

(3) The following deduction would be valid were cyclic kindings allowed.

$$\dfrac{\{t\colon \langle l\colon t\rangle\} \vdash t \blacktriangleright \langle l\colon t\rangle}{\{t\colon \langle l\colon t\rangle\} \vdash a\lhd l\colon a \blacktriangleright \{a\colon t\}}$$

DEFINITION 5.6 (Substitution on types). A *substitution on types* is a total mapping $s : \mathbf{V} \to \mathbf{T}_{[]}$ from type-variables to types. Such a substitution can be extended to types α in $\mathbf{T}_{[]}$ and kinds k in $\mathbf{K}$, by substituting the type-variables in α and in k, and to typings Γ and kindings K by substituting the type-variables in the predicates of Γ and K (and leaving the subjects unchanged).

Compatibility of typings is preserved by substitution.

PROPOSITION 5.7. *Let s be a substitution. If $\Gamma \asymp \Delta$, then $s\Gamma \asymp s\Delta$.*

PROOF. Directly from the definition of typing compatibility. □

System $\mathrm{TA}_{[]\langle\rangle}$ is correct with respect to system $\mathrm{TA}_{[]}$ in the sense that terms typable in one system are typable in the other.

THEOREM 5.8. 1. *$\vdash_{[]} P \blacktriangleright \Gamma$ if and only if $\vdash_{[]\langle\rangle} P \blacktriangleright \Gamma$.*
2. *If $K \vdash_{[]\langle\rangle} P \blacktriangleright \Gamma$, then there is a substitution s such that $\vdash_{[]} P \blacktriangleright s\Gamma$.*

PROOF. (1) ($\Rightarrow$) Suppose that $\vdash_{[]} P \blacktriangleright \Gamma$. Replace, throughout the deduction of $\vdash_{[]} P \blacktriangleright \Gamma$, occurrences of the MSG-axiom

$$a \lhd l_1 : \tilde{v} \blacktriangleright \{a\colon [l_1\colon \tilde{\alpha}_1, \ldots, l_n\colon \tilde{\alpha}_n], \tilde{v}\colon \tilde{\alpha}_1\}$$

by the deduction (composed of the KRCD-axiom and the MSG$_k$-rule)

$$\frac{[l_1\colon \tilde{\alpha}_1, \ldots, l_n\colon \tilde{\alpha}_n] \blacktriangleright \langle l_1\colon \tilde{\alpha}_1\rangle}{a \lhd l_1 : \tilde{v} \blacktriangleright \{a\colon [l_1\colon \tilde{\alpha}_1, \ldots, l_n\colon \tilde{\alpha}_n], \tilde{v}\colon \tilde{\alpha}_1\}}$$

to obtain a deduction of $\vdash_{[]\langle\rangle} P \blacktriangleright \Gamma$.
($\Leftarrow$) By the second part of this lemma.

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

Noting that typing compatibility is preserved by substitution (proposition 5.7), replace throughout the deduction of $K \vdash_{[]\langle\rangle} P \blacktriangleright \Gamma$, 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 $\vdash_{[]} P \blacktriangleright s\Gamma$.

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

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

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

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

3. Kinded Recursive Typing Assignment System $\mathbf{TA}_{[]\mu\langle\rangle}$

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 explicit recursive kind expressions. Although predicates in typings remain kinds in $\mathbf{K}$, the KRCD-axiom in definition 5.3 may now assign a kind with recursive types to a record in $\mathbf{T}_{[]\mu}$.

DEFINITION 5.9 (Kinded recursive typing assignment system $\mathrm{TA}_{[]\mu\langle\rangle}$). The kinded recursive typing assignment system $\mathrm{TA}_{[]\mu\langle\rangle}$ is composed of the axioms and rules in the definition 5.4 of the kinded typing typing assignment system $\mathrm{TA}_{[]\langle\rangle}$, with the MSG_k-rule replaced by the $\mathrm{MSG}_{\mu\langle\rangle}$-rule below, by adding the $\approx$-rule in the definition 4.14 of the recursive type system $\mathrm{TA}_{[]\mu}$ with formulas of the form $P \blacktriangleright \Gamma$ replaced by $K \vdash P \blacktriangleright \Gamma$, and by allowing cycles in kindings.

($\mathrm{MSG}_{\mu\langle\rangle}$)

$$\frac{K \vdash \beta \blacktriangleright \langle l\colon \tilde{\alpha}\rangle}{K \vdash a \lhd l\colon\tilde{v} \blacktriangleright \{a\colon \gamma, \tilde{v}\colon \tilde{\alpha}\}} \qquad \text{(condition: } \beta \approx \gamma)$$

Write $K \vdash_{[]\mu\langle\rangle} P \blacktriangleright \Gamma$ if there is a deduction of $K \vdash P \blacktriangleright \Gamma$ in $\mathrm{TA}_{[]\mu\langle\rangle}$, for K a possibly cyclic kinding.

REMARK 5.10. It is also possible to define a kinded recursive type system using the MSG_k-rule instead of the $\mathrm{MSG}_{\mu\langle\rangle}$-rule. The new rule, however allows to type messages whose target is contained in the message's communication (cf. remark 4.15). For example, using the KRCD-axiom, and the $\mathrm{MSG}_{\mu\langle\rangle}$-rule, one has the deduction

$$\frac{[l\colon \mu t.[l\colon t]] \blacktriangleright \langle l\colon \mu t.[l\colon t]\rangle}{a \lhd l\colon a \blacktriangleright \{a\colon \mu t.[l\colon t]\}}$$

since $[l\colon \mu t.[l\colon t]] \approx \mu t.[l\colon t]$. Observe also that, when β is a type-variable, the new $\mathrm{MSG}_{\mu\langle\rangle}$-rule reduces to the old MSG_k-rule.

EXAMPLE 5.11. Let K be the cyclic typing $\{u\colon \langle cons\colon t\,[val\colon u]\rangle\}$. One has the following deduction for term Cons cvc, starting from the KVAR-axiom and applying the $\mathrm{MSG}_{\mu\langle\rangle}$ and the OBJ_k-rule.

$$\cfrac{\cfrac{K \vdash u \blacktriangleright \langle cons\colon t\,[val\colon u]\rangle}{K \vdash r \lhd cons\colon vc \blacktriangleright \{v\colon t, c\colon [val\colon u], r\colon u\}}}{K \vdash c \rhd [val\colon (r)\;\; r \lhd cons\colon vc] \blacktriangleright \{v\colon t, c\colon [val\colon u]\}}$$

In this case, since name r (representing the address of the object meant to receive the reply $cons : vc$) 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

$$\mathrm{Cons}\, cvc,\ \nu r(c \lhd \mathsf{val}{:}r,\ r \rhd [nil{:}P\ \ \&\ \ cons{:}(xy)\, Q])$$

where the Cons cell is evaluated, and the computation branches, according to the result, into terms P or Q. Here, the principal type of c is a recursive type $\alpha = \mu u.[\mathsf{val}\colon [cons\colon tu, nil]]$. The derivation of the subterm $\mathrm{Cons}\, cvc$ is depicted below, starting with the KRCD-axiom, and applying the MSG_k, the $\approx$, the OBJ_k, and the $\approx$-rule in succession. Notice that $\alpha \approx [\mathsf{val}\colon [cons\colon t\alpha, nil]]$.

$$\dfrac{\dfrac{\dfrac{\dfrac{[cons\colon t\alpha, nil] \blacktriangleright \langle cons\colon t\alpha\rangle}{r \lhd cons{:}vc \blacktriangleright \{v\colon t, c\colon \alpha, r\colon [cons\colon t\alpha, nil]\}}}{r \lhd cons{:}vc \blacktriangleright \{v\colon t, c\colon [\mathsf{val}\colon [cons\colon t\alpha, nil]], r\colon [cons\colon t\alpha, nil]\}}}{c \rhd [\mathsf{val}{:}(r)\ r \lhd cons\colon vc] \blacktriangleright \{v\colon t, c\colon [\mathsf{val}\colon [cons\colon t\alpha, nil]]\}}}{c \rhd [\mathsf{val}{:}(r)\ r \lhd cons\colon vc] \blacktriangleright \{v\colon t, c\colon \alpha\}}$$

System $\mathrm{TA}_{[]\mu\langle\rangle}$ is correct with respect to system $\mathrm{TA}_{[]\mu}$ in the same sense that system $\mathrm{TA}_{[]\langle\rangle}$ is correct with respect to system $\mathrm{TA}_{[]}$, that is, in that terms typable in one system are typable in the other.

THEOREM 5.12. 1. $\vdash_{[]\mu} P \blacktriangleright \Gamma$ *if and only if* $\vdash_{[]\mu\langle\rangle} P \blacktriangleright \Gamma$.
2. *If* $K \vdash_{[]\mu\langle\rangle} P \blacktriangleright \Gamma$*, then there is a substitution* s *such that* $\vdash_{[]\mu} P \blacktriangleright s\Gamma$.

PROOF. The proof of the corresponding result in systems $\mathrm{TA}_{[]}$ and $\mathrm{TA}_{[]\langle\rangle}$, theorem 5.8, easily adapts to the recursive setting. The key issues are that type equivalence is preserved by substitution, and that the system of equations generated by a (possibly cyclic) kinding has a unique solution whose components are regular trees, and that each of these trees define a type in $\mathbf{T}_{[]\mu}$. The cases concerning the MSG_μ-axiom and the $\text{MSG}_{\mu\langle\rangle}$-rule also need a slight adaptation. The details are explicated below.

(1) ($\Rightarrow$) Suppose that $\vdash_{[]\mu} P \blacktriangleright \Gamma$. Replace, throughout the deduction of $\vdash_{[]\mu} P \blacktriangleright \Gamma$, occurrences of the MSG_μ-axiom

$$a \lhd l_1 {:} \tilde{v} \blacktriangleright a\colon \gamma \cup \tilde{v}\colon \tilde{\alpha}_1 \qquad (\gamma \approx [l_1\colon \tilde{\alpha}_1, \ldots, l_n\colon \tilde{\alpha}_n])$$

by the deduction (composed of the KRCD-axiom and the $\mathrm{MSG}_{\mu\langle\rangle}$-rule)

$$\frac{[l_1\colon \tilde{\alpha}_1, \ldots, l_n\colon \tilde{\alpha}_n] \blacktriangleright \langle l_1\colon \tilde{\alpha}_1\rangle}{a \lhd l_1 \colon \tilde{v} \blacktriangleright \{a\colon \gamma, \tilde{v}\colon \tilde{\alpha}_1\}} \qquad (\gamma \approx [l_1\colon \tilde{\alpha}_1, \ldots, l_n\colon \tilde{\alpha}_n])$$

to obtain a deduction of $\vdash_{[]\mu\langle\rangle} P \blacktriangleright \Gamma$.

($\Leftarrow$) By the second part of this lemma.

(2) Suppose that $K \vdash_{[]\mu\langle\rangle} P \blacktriangleright \Gamma$. I build a substitution s from K, and show that $\vdash_{[]\mu} P \blacktriangleright s\Gamma$. Without loss of generality, assume that the subjects of K are $t_1, \ldots, t_n$. Define a system of equations S with unknowns $t_1, \ldots, t_n$, and equations of the form

$$t_i = l_{i_1} \cdots l_{i_k}(\tilde{\alpha}^*_{i_1}, \ldots, \tilde{\alpha}^*_{i_k})$$

for each kind assignment $t_i\colon \langle l_{i_1}\colon \tilde{\alpha}_{i_1}, \ldots, l_{i_k}\colon \tilde{\alpha}_{i_k}\rangle$ in K. Clearly, the right-hand sides of the equations are non-variable regular trees, and hence S is a generalized regular Greibach system with unknowns $t_1, \ldots, t_n$. As such, S has a unique solution $(\rho_1, \ldots, \rho_n)$ in $\mathbf{T}^n_{\mathrm{R}}$ (Courcelle [**18**], theorem 4.3.1). By lemma 4.11, each of these trees define a type in $\mathbf{T}_{[]\mu}$. Let β_i be a type in $\mathbf{T}_{[]\mu}$ such that $\beta^*_i = \rho_i$, for each $i = 1, \ldots n$. Define a substitution $s\colon \mathbf{V} \to \mathbf{T}_{[]\mu}$ by putting $st_i = \beta_i$ if $1 \leq i \leq n$, and $st = t$ otherwise.

Noting that both typing compatibility and type equivalence are preserved by substitution (propositions 5.7 and 4.13), replace throughout the deduction of $K \vdash_{[]\langle\rangle} P \blacktriangleright \Gamma$, 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 $\vdash_{[]\mu} P \blacktriangleright s\Gamma$.

$$\frac{K \vdash [l_1\colon \tilde{\alpha}_1, \ldots, l_n\colon \tilde{\alpha}_n] \blacktriangleright \langle l_1\colon \tilde{\alpha}_1\rangle}{K \vdash a \lhd l_1 \colon \tilde{v} \blacktriangleright \{a\colon \gamma, \tilde{v}\colon \tilde{\alpha}_1\}} \quad (\gamma \approx [l_1\colon \tilde{\alpha}_1, \ldots, l_n\colon \tilde{\alpha}_n])$$

$$a \lhd l_1 \colon \tilde{v} \blacktriangleright \{a\colon s\gamma, \tilde{v}\colon s\tilde{\alpha}_1\} \quad (s\gamma \approx s[l_1\colon \tilde{\alpha}_1, \ldots, l_n\colon \tilde{\alpha}_n])$$

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

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

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

$$(\alpha \approx \beta) \qquad\qquad (s\alpha \approx a\beta) \quad \square$$

4. Subsumption and Principal Typings

DEFINITION 5.13 (Subsumption).

1. A *kinded substitution* is a pair (K, s) composed of a kinding K and a substitution s.
2. (K', s) *respects* K if $K' \vdash_{\langle\rangle} st \blacktriangleright sKt$, for all $t \in \operatorname{dom} K$.
3. A *kinded typing* is a pair (K, Γ) composed of a kinding K and a typing Γ.
4. (K, Γ) *subsumes* (K', Δ) if $\operatorname{dom} K \subseteq \operatorname{dom} K'$, and there is a substitution s such that (K', s) respects K, and $s\Gamma \subseteq \Delta$. When (K, Γ) subsumes (K', Δ), I also say that (K', Δ) is an *instance* of (K, Γ).

LEMMA 5.14. *Subsumption is decidable.*

PROOF. It is easy to see that (K, Γ) subsumes (K', Δ) if and only if

1. $\operatorname{dom} K \subseteq \operatorname{dom} K'$, and-then
2. $\operatorname{dom} \Gamma \subseteq \operatorname{dom} \Delta$, and-then
3. $\mathcal{U}\{\Gamma x \doteq \Delta x \mid x \in \operatorname{dom} \Gamma\}$ succeeds with a mgu s, and-then
4. $s\Gamma = \Gamma$, and-then
5. (K', s) respects K,

where $\mathcal{U}$ is the unification algorithm 5.21 or 6.8. Clearly, $\Gamma = s\Gamma = s\Delta \restriction (\operatorname{dom} \Gamma)$. The 'respects' predicate is decidable since $\operatorname{dom} K$ is finite, and $\vdash_{\langle\rangle}$ has only axioms. □

Instances of well-typings are also well-typings.

LEMMA 5.15 (Subsumption lemma in $\mathrm{TA}_{[]\langle\rangle}$). *If* $K \vdash_{[]\langle\rangle} P \blacktriangleright \Gamma$ *and* (K, Γ) *subsumes* (K', Δ)*, then* $K' \vdash_{[]\langle\rangle} P \blacktriangleright \Delta$.

PROOF. Let s be a substitution such that $K' \vdash_{\langle\rangle} st \blacktriangleright sKt$ for all $t \in \operatorname{dom} K$. I claim that

$$K' \vdash_{[]\langle\rangle} P \blacktriangleright s\Gamma,$$

and use the WEAK-rule as many times as needed to obtain $K' \vdash_{[]\langle\rangle} P \blacktriangleright \Delta$.

To prove the claim, replace throughout the deduction of $K \vdash_{[]\langle\rangle} P \blacktriangleright \Gamma$, occurrences of the MSG_k-rule, formulas and side conditions below on the left by the corresponding ones on the right, to obtain a derivation of $K' \vdash_{[]\langle\rangle} P \blacktriangleright s\Gamma$.

Notice that typing compatibility is preserved by substitution (proposition 5.7).

$$\frac{K \vdash [l_1\colon \tilde{\alpha}_1, \ldots, l_n\colon \tilde{\alpha}_n] \blacktriangleright \langle l_1\colon \tilde{\alpha}_1\rangle}{K \vdash a \lhd l_1 \colon \tilde{v} \blacktriangleright \{a\colon [l_1\colon \tilde{\alpha}_1, \ldots, l_n\colon \tilde{\alpha}_n], \tilde{v}\colon \tilde{\alpha}_1\}}$$

$$\frac{K' \vdash s[l_1\colon \tilde{\alpha}_1, \ldots, l_n\colon \tilde{\alpha}_n] \blacktriangleright s\langle l_1\colon \tilde{\alpha}_1\rangle}{K' \vdash a \lhd l_1 \colon \tilde{v} \blacktriangleright \{a\colon s[l_1\colon \tilde{\alpha}_1, \ldots, l_n\colon \tilde{\alpha}_n], \tilde{v}\colon s\tilde{\alpha}_1\}}$$

$$\frac{K \vdash t \blacktriangleright \langle l_1\colon \tilde{\alpha}_1\rangle}{K \vdash a \lhd l_1 \colon \tilde{v} \blacktriangleright \{a\colon t, \tilde{v}\colon \tilde{\alpha}_1\}} \qquad \frac{K' \vdash st \blacktriangleright s\langle l_1\colon \tilde{\alpha}_1\rangle}{K' \vdash a \lhd l_1 \colon \tilde{v} \blacktriangleright \{a\colon st, \tilde{v}\colon s\tilde{\alpha}_1\}}$$

$$K \vdash P \blacktriangleright \Gamma \qquad\qquad K' \vdash P \blacktriangleright s\Gamma$$

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

LEMMA 5.16 (Subsumption lemma in $\mathrm{TA}_{[]\mu\langle\rangle}$). *If* $K \vdash_{[]\mu\langle\rangle} P \blacktriangleright \Gamma$ *and* (K,Γ) *subsumes* (K',Δ), *then* $K' \vdash_{[]\mu\langle\rangle} P \blacktriangleright \Delta$.

PROOF. The proof of the corresponding result in $\mathrm{TA}_{[]\langle\rangle}$ (lemma 5.15) extends to the system $\mathrm{TA}_{[]\mu\langle\rangle}$ with few changes. The proof of the claim that $K' \vdash_{[]\mu\langle\rangle} P \blacktriangleright s\Gamma$ easily accommodates the $\approx$-rule. For the new $\mathrm{MSG}_{\mu\langle\rangle}$-rule, the only case that changes is when deductions start with the KRCD-axiom. Noticing that type equivalence is preserved by substitution (proposition 4.13), replace occurrences of deductions of the form

$$\frac{K \vdash [l_1\colon \tilde{\alpha}_1, \ldots, l_n\colon \tilde{\alpha}_n] \blacktriangleright \langle l\colon \tilde{\alpha}_1\rangle}{K \vdash a \lhd l \colon \tilde{v} \blacktriangleright \{a\colon \gamma, \tilde{v}\colon \tilde{\alpha}_1\}} \qquad (\gamma \approx [l_1\colon \tilde{\alpha}_1, \ldots, l_n\colon \tilde{\alpha}_n])$$

by deductions of the form

$$\frac{K' \vdash s[l_1\colon \tilde{\alpha}_1, \ldots, l_n\colon \tilde{\alpha}_n] \blacktriangleright \langle sl\colon \tilde{\alpha}_1\rangle}{K' \vdash a \lhd l \colon \tilde{v} \blacktriangleright \{a\colon s\gamma, \tilde{v}\colon s\tilde{\alpha}_1\}} \qquad (s\gamma \approx s[l_1\colon \tilde{\alpha}_1, \ldots, l_n\colon \tilde{\alpha}_n]) \quad \square$$

DEFINITION 5.17 (Principal kinded-typings). A principal kinded-typing for P is a well kinded-typing for P that subsumes all other well kinded-typings for P. In other words, (K,Γ) is principal for P in system $\mathrm{TA}_{[]\langle\rangle}$ if

1. $K \vdash_{[]\langle\rangle} P \blacktriangleright \Gamma$, and
2. if $K' \vdash_{[]\langle\rangle} P \blacktriangleright \Delta$, then (K,Γ) subsumes (K',Δ).

Principal kinded-typings in system $\mathrm{TA}_{[]\mu\langle\rangle}$ are defined as above with '$\vdash_{[]\langle\rangle}$' replaced by '$\vdash_{[]\mu\langle\rangle}$'.

Proposition 5.18. *Principal kinded-typings, when they exist, are unique up to renaming of type-variables, and contain exactly the free names in the process.*

Proof. Consider two principal kinded-typings (K, Γ) and (K', Δ). Since (K, Γ) is principal, $\operatorname{dom} K \subseteq \operatorname{dom} K'$, and there is a substitution s such that $s\Gamma \subseteq \Delta$, and $K' \vdash_{\langle\rangle} st \blacktriangleright sKt$ for all $t \in \operatorname{dom} K$. Since (K', Δ) is also principal, $\operatorname{dom} K' \subseteq \operatorname{dom} K$, and there is a substitution s' such that $s'\Delta \subseteq \Gamma$, and $K \vdash_{\langle\rangle} s't \blacktriangleright s'Kt$ for all $t \in \operatorname{dom} K'$. Then $s's\Gamma \subseteq s'\Delta \subseteq \Gamma$ and $ss'\Delta \subseteq s\Gamma \subseteq \Delta$, and the type-variables in Γ and Δ are in a bijection. Further, $\operatorname{dom} K = \operatorname{dom} K'$, and one has $K' \vdash_{\langle\rangle} st \blacktriangleright sKt$ and $K \vdash_{\langle\rangle} s't \blacktriangleright s'K't$ for all $t \in \operatorname{dom} K$.

For the second part of the lemma, let P be a term and (K, Γ) its principal kinded-typing. That Γ contains at least the free names of P follows by an adaptation of the typing lemma 3.13(1). To see that Γ contains no more names than those free in P, suppose that $(K, \Gamma \cdot x\colon \alpha)$ is also principal for P. Clearly this typing is an instance of (K, Γ) and hence it is not principal for P. $\square$

Existence and computability of principal typings is the main property of the kinded type systems $\mathrm{TA}_{[]\langle\rangle}$ and $\mathrm{TA}_{[]\mu\langle\rangle}$; its proof appears at the end of section 6.

Theorem 5.19 (Existence and computability of principal typings). *If term P is typable in $TA_{[]\langle\rangle}$ (resp. in $TA_{[]\mu\langle\rangle}$), there exists a principal kinded-typing for P in $\mathbf{T}_{[]}$ (resp. in $\mathbf{T}_{[]\mu}$). It can be effectively computed.*

5. Typing Inference

Definition 5.20 (Unifier).

1. A *kinded set of equations* is a pair (K, E) composed of a kinding K, and a set of equations E of the form $\alpha \doteq \beta$, for α and β types.
2. (K', s) is a *unifier* of (K, E) if (K', s) respects K, and $s\alpha = s\beta$ for all $\alpha \doteq \beta \in E$.
3. (K, s) is *more general than* (K', s') if there is a substitution s'' such that $s' = s''s$ and (K', s'') respects K.

Theorem 5.21 (Kinded unification in $\mathbf{T}_{[]}$, Ohori [**43**] theorem 2). *There is an algorithm that, given any kinded set of equations in $TA_{[]\langle\rangle}$, computes a most general unifier if it exists, and reports failure otherwise.*

Chapter 6 deals with unification of kinded sets of equations in $\mathrm{TA}_{[]\mu\langle\rangle}$. In particular, given any kinded set of equations in $\mathrm{TA}_{[]\mu\langle\rangle}$, algorithm 6.16 computes a most general unifier if it exists, and reports failure otherwise.

NOTATION 5.22. Let $\tilde{x} = x_1 \dots x_n$, for some $n \geq 0$. Then, $\Gamma\tilde{x}$ denotes the sequence of types $\Gamma x_1 \dots \Gamma x_n$.

The typing inference algorithm builds, from a process P_0 with bound names renamed to be distinct from the free names, a typing and a kinded set of equations. Suppose the algorithm produces a typing Γ_0 and a kinded set of equations (K, E). 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 $s\Gamma_0 \restriction \mathrm{fn}\, P_0$, for (K', s) a unifier of (K, E).

ALGORITHM 5.23.

Input: A term P_0 with all bound names renamed to be distinct.

Initialization: Set K and E to $\emptyset$, G to $\{P_0\}$, and Γ_0 to $\{x_1\colon t_1, \dots, x_n\colon t_n\}$, for $x_1, \dots, x_n$ the names in P_0, and $t_1, \dots, t_n$ pairwise distinct type-variables.

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

- **Case P is $a \triangleleft l{:}\tilde{v}$:** Generate the equation $\Gamma_0 a \doteq t$ and the kind assignment $t\colon \langle l\colon \Gamma_0\tilde{v}\rangle$, for t a fresh type-variable.
- **Case P is $a \triangleright [l_1 : (\tilde{x}_1)\, P_1 \,\&\, \cdots \,\&\, l_n : (\tilde{x}_n)\, P_n]$:** Generate the equation $\Gamma_0 a \doteq [l_1\colon \Gamma_0\tilde{x}_1, \dots, l_n\colon \Gamma_0\tilde{x}_n]$ and the goals $P_1, \dots, P_n$.
- **Case P is Q, R:** Generate the goals Q and R.
- **Case P is $\nu x Q$ or $!Q$:** Generate the goal Q.

Output: (Γ_0, K, E).

To build the principal kinded-typing of a term P_0, use the algorithm above on P_0, and then the kinded unification algorithm on the resulting kinded set of equations. If the kinded set of equations has no solution, then P_0 is not typable. Otherwise let (K, s) be its most general unifier. Then, $(K, s\Gamma_0 \restriction \mathrm{fn}\, P_0)$ is the principal kinded-typing of P_0. It follows by lemma 5.15 that every instance of $(K, s\Gamma_0 \restriction \mathrm{fn}\, P_0)$ is a well kinded-typing for P_0. The proof of the below lemma is postponed until the end of section 6.

LEMMA 5.24. *Let (Γ_0, K, E) be the result of algorithm 5.23 on the input P_0.*

1. *If (K', s) is the most general unifier of (K, E), then $(K', s\Gamma_0 \restriction \mathrm{fn}\, P_0)$ is principal for P_0.*
2. *If (K, E) is not unifiable, then P is not typable.*

EXAMPLE 5.25. Consider the buffer cell described in example 2.5(5). To conform to the constraint on input terms, rename bound names s and v, to obtain the term

$$\nu x(x \lhd \mathit{recur}\colon sv,$$
$$!x \rhd [\mathit{recur}\colon (s'v')\, s' \rhd [\mathit{read}\colon (r)\, r \lhd \mathrm{val}\colon v', x \lhd \mathit{recur}\colon s'v' \& \mathit{write}\colon (u)\, x \lhd \mathit{recur}\colon s'u]]).$$

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

$\{P_0\}$			Initial state
$\{x \lhd \mathit{recur}\colon sv, !x \rhd [\cdots]\}$			Scope
$\{x \lhd \mathit{recur}\colon sv; !x \rhd [\cdots]\}$			Composition
$\{!x \rhd [\mathit{recur}\colon (s'v')\ \cdots]\}$	$t_x \doteq t_1$	$t_1\colon \langle \mathit{recur}\colon t_s t_v\rangle$	Message
$\{x \rhd [\mathit{recur}\colon (s'v')\ \cdots]\}$			Replication
$\{s' \rhd [\mathit{read}\colon \cdots\ \&\ \mathit{write}\colon \cdots]\}$	$t_x \doteq [\mathit{recur}\colon t'_s t'_v]$		Object
$\{r \lhd \mathrm{val}\colon v', x \lhd \mathit{recur}\colon s'v'; \ldots\}$	$t'_s \doteq [\mathit{read}\colon t_r, \mathit{write}\colon t_u]$		Object
$\{r \lhd \mathrm{val}\colon v'; x \lhd \mathit{recur}\colon s'v'; \ldots\}$			Composition
$\{x \lhd \mathit{recur}\colon s'v'; x \lhd \mathit{recur}\colon s'u\}$	$t_r \doteq t_2$	$t_2\colon \langle \mathrm{val}\colon t'_v\rangle$	Message
$\{x \lhd \mathit{recur}\colon s'u\}$	$t_x \doteq t_3$	$t_3\colon \langle \mathit{recur}\colon t'_s t'_v\rangle$	Message
$\{\}$	$t_x \doteq t_4$	$t_4\colon \langle \mathit{recur}\colon t'_s t_u\rangle$	Message

The resulting kinded set of equations is composed of the kinding

$$\begin{aligned}
\{&t_1\colon \langle \mathit{recur}\colon t_s t_v\rangle,\\
&t_2\colon \langle \mathrm{val}\colon t'_v\rangle,\\
&t_3\colon \langle \mathit{recur}\colon t'_s t'_v\rangle,\\
&t_4\colon \langle \mathit{recur}\colon t'_s t_u\rangle\},
\end{aligned}$$

and the set of equations

$$\begin{aligned}
\{&t_x \doteq t_1,\\
&t_x \doteq [\mathit{recur}\colon t'_s t'_v],\\
&t'_s \doteq [\mathit{read}\colon t_r, \mathit{write}\colon t_u],\\
&t_r \doteq t_2,\\
&t_x \doteq t_3,\\
&t_x \doteq t_4\}.
\end{aligned}$$

Solving this kinded set of equations using a kinded unification algorithm yields the kinded substitution composed of kinding

$$K = \{t_r\colon \langle \mathrm{val}\colon t_v\rangle\},$$

and substitution

$$\begin{aligned}
s = \{&t_1 \mapsto t_3 \mapsto t_4 \mapsto t_x \mapsto [\mathit{recur}\colon [\mathit{read}\colon t_r, \mathit{write}\colon t_v] t_v],\\
&t_s \mapsto t'_s \mapsto [\mathit{read}\colon t_r, \mathit{write}\colon t_v],\\
&t'_v \mapsto t_v \mapsto t_u,\\
&t_2 \mapsto t_r\}.
\end{aligned}$$

Since the free names of P_0 are v and c, the principal kinded-typing $(K, s\Gamma_0 \upharpoonright \mathrm{fn}\, P_0)$ for P_0 is

$$(\{t_r\colon \langle \mathrm{val}\colon t_v\rangle\}, \{v\colon t_v, c\colon [\mathit{read}\colon t_r, \mathit{write}\colon t_v]\}).$$

6. Correctness of the Algorithm

THEOREM 5.26 (Termination). *Algorithm 5.23 always terminates.*

PROOF. Each step generates subgoals involving terms strictly smaller than the original. □

The following notation simplifies the formulation of the invariant of the algorithm.

NOTATION 5.27. Let Γ_0 be a typing, and G a set of terms. A kinded-substitution (K', s) *solves* (K, E, G) if (K', s) unifies (K, E), and $K' \vdash_{[]\langle\rangle} P \blacktriangleright s\Gamma_0$ for each term P in G.

INVARIANT 5.28.

(Soundness)

$$\forall (K', s) : (K', s) \text{ solves } (K, E, G) \;\Rightarrow\; K' \vdash_{[]\langle\rangle} P_0 \blacktriangleright s\Gamma_0.$$

(Completeness)

$$K \vdash_{[]\langle\rangle} P_0 \blacktriangleright \Gamma \;\Rightarrow\; \exists (K', s) : (K', s) \text{ solves } (K, E, G) \wedge \Gamma \restriction \mathrm{fn}\, P_0 = s\Gamma_0 \restriction \mathrm{fn}\, P_0.$$

At termination, when $G = \emptyset$, one has

$$\forall (K', s) : (K', s) \text{ unifies } (K, E) \;\Rightarrow\; K' \vdash_{[]\langle\rangle} P_0 \blacktriangleright s\Gamma_0,$$

$$K \vdash_{[]\langle\rangle} P_0 \blacktriangleright \Gamma \;\Rightarrow\; \exists (K', s) : (K', s) \text{ unifies } (K, E) \wedge \Gamma \restriction \mathrm{fn}\, P_0 = s\Gamma_0 \restriction \mathrm{fn}\, P_0,$$

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

THEOREM 5.29 (Correctness of the algorithm). *The invariants are established by the initialization step and preserved by each case in the loop.*

PROOF. (Soundness) The initialization step is trivial. The proof that the cases in the loop preserve the soundness invariant is by induction on the structure of P_0.

Case P_0 is $\nu x P$. Take a (K', s) that solves $(K, E, G \cup \{P\})$. By induction $K' \vdash_{[]\langle\rangle} P \blacktriangleright s\Gamma_0$; the result follows by the SCOP-rule followed by the WEAK-rule.

Case P_0 is P, Q. Take a (K', s) that solves $(K, E, G \cup \{(P, Q)\})$. By induction $K' \vdash_{[]\langle\rangle} P \blacktriangleright s\Gamma_0$, and $K' \vdash_{[]\langle\rangle} Q \blacktriangleright s\Gamma_0$. Since $s\Gamma_0 \asymp s\Gamma_0$, the result follows by the COMP-rule.

Case P_0 is $a \lhd l : \tilde{v}$. Take a (K', s) that solves $(K \cup \{t \colon \langle l \colon \Gamma_0\tilde{v}\rangle\}, E \cup \{t \doteq \Gamma_0 a\}, G)$. By induction $K' \vdash_{\langle\rangle} st \blacktriangleright s\langle l \colon \Gamma_0\tilde{v}\rangle$, and $s\Gamma_0 a = st$. By the MSG$_k$-rule, $K' \vdash_{[]\langle\rangle} a \lhd l : \tilde{v} \blacktriangleright \{a \colon st, \tilde{v} \colon s\Gamma_0\tilde{v}\}$. Since $s\Gamma_0 a = st$, $K' \vdash_{[]\langle\rangle} a \lhd l : \tilde{v} \blacktriangleright s\{a \colon t, \tilde{v} \colon \Gamma_0\tilde{v}\}$. Use the WEAK-rule as many times as needed to obtain the result.

Case P_0 is $a \rhd [l_1 : (\tilde{x}_1)\, P_1 \;\&\; \cdots \;\&\; l_n : (\tilde{x}_n)\, P_n]$. Take a (K', s) that solves $(K, E \cup \{\Gamma_0 a \doteq [l_1 \colon \Gamma_0\tilde{x}_1, \ldots, l_n \colon \Gamma_0\tilde{x}_n]\}, G \cup \{P_1, \ldots, P_n\})$. By induction $s\Gamma_0 a = s[l_1 \colon \Gamma_0\tilde{x}_1, \ldots, l_n \colon \Gamma_0\tilde{x}_n]$, and $K' \vdash_{[]\langle\rangle} P_i \blacktriangleright s\Gamma_0$ for $1 \le i \le n$. Since $s\Gamma_0 \asymp s\Gamma_0$, one has $s\Gamma_0/\tilde{x}_1 \asymp \cdots \asymp s\Gamma_0/\tilde{x}_n$. By the OBJ-rule,

$$K' \vdash_{[]\langle\rangle} a \rhd [l_1 : (\tilde{x}_1)\, P_1 \;\&\; \cdots \;\&\; l_n : (\tilde{x}_n)\, P_n] \blacktriangleright s\Gamma_0/\tilde{x}_1 \ldots \tilde{x}_n.$$

Use the WEAK-rule as many times as needed to obtain the result.

(Completeness) Suppose $K \vdash_{[]\langle\rangle} P_0 \blacktriangleright \Gamma$. For the initialization step, define a substitution s by putting

$$st = \begin{cases} \Gamma x & \text{if } t = \Gamma_0 x \text{ for some } x \in \operatorname{fn} P_0, \\ t & \text{otherwise.} \end{cases}$$

Using the kinded substitution (K, s), I only need to show that $K \vdash_{[]\langle\rangle} P_0 \blacktriangleright s\Gamma_0$. By hypothesis and the typing lemma 3.13, $K \vdash_{[]\langle\rangle} P_0 \blacktriangleright \Gamma \restriction \operatorname{fn} P_0$. By the definition of s, $K \vdash_{[]\langle\rangle} P_0 \blacktriangleright s(\Gamma_0 \restriction \operatorname{fn} P_0)$. Use the WEAK-rule as many times as needed to obtain the result.

For each of the cases analysed in the loop step, by hypothesis, there is a kinded substitution (K', s) such that (K', s) solves (K, E, G) and $\Gamma \restriction \operatorname{fn} P_0 = s(\Gamma_0 \restriction \operatorname{fn} P_0)$. I build a substitution s', and show that the completeness invariant holds for (K', s') after the step. I start with the easy cases, and progress towards the more complex ones. The case when goal chosen is a replicated process is trivial.

Case the goal chosen is $\nu x P$. Using the kinded substitution (K', s), I show that $K' \vdash_{[]\langle\rangle} P \blacktriangleright s\Gamma_0$. By induction, $K' \vdash_{[]\langle\rangle} \nu x P \blacktriangleright s\Gamma_0$, and by the SCOP-rule, $K' \vdash_{[]\langle\rangle} \nu x P \blacktriangleright s\Gamma_0/x$. The result follows by the WEAK-rule.

Case the goal chosen is P, Q. Using the kinded substitution (K', s), I show that $K' \vdash_{[]\langle\rangle} P \blacktriangleright s\Gamma_0$ and $K' \vdash_{[]\langle\rangle} Q \blacktriangleright s\Gamma_0$. By induction, $K' \vdash_{[]\langle\rangle} P, Q \blacktriangleright s\Gamma_0$.

By the COMP-rule, $K' \vdash_{[]\langle\rangle} P \blacktriangleright \Gamma$, $K' \vdash_{[]\langle\rangle} Q \blacktriangleright \Gamma$, for some Γ, Δ such that $\Gamma \cup \Delta = s\Gamma_0$. In each case, the result follows by a series of applications of the WEAK-rule.

Case the goal chosen is $a \lhd l : \tilde{v}$. Define a substitution s' by putting

$$s't' = \begin{cases} s\Gamma_0 a & \text{if } t = t', \\ st' & \text{otherwise,} \end{cases}$$

where t is the fresh variable introduced in the loop step. Using the kinded substitution (K', s'), and since t is fresh, only the new kind assignment and equation need consideration. I show that $K' \vdash_{\langle\rangle} s't \blacktriangleright s'\langle l \colon \Gamma_0 \tilde{v}\rangle$ and $s'\Gamma_0 a = s't$. For the first result, by hypothesis, one has $K' \vdash_{[]\langle\rangle} a \lhd l : \tilde{v} \blacktriangleright s\Gamma_0$; by the MSG$_k$-rule, $K' \vdash_{\langle\rangle} s\Gamma_0 a \blacktriangleright \langle l \colon s\Gamma_0 \tilde{v}\rangle$; and by the definition of s', $K' \vdash_{\langle\rangle} s't \blacktriangleright s'\langle l \colon s\Gamma_0 \tilde{v}\rangle$. The second result follows directly from the definition of s'.

Case the goal chosen is $a \rhd [l_1 : (\tilde{x}_1)\, P_1 \,\&\cdots\&\, l_n : (\tilde{x}_n)\, P_n]$. Define a substitution s' by putting

$$s'\tilde{t} = \begin{cases} \tilde{\alpha}_i & \text{if } \tilde{t} = \Gamma_0 \tilde{x}_i, \text{ for } 1 \leq i \leq n, \\ s\tilde{t} & \text{otherwise.} \end{cases}$$

Since the bound names of P_0 are distinct from the free names, names $\tilde{x}_1, \ldots \tilde{n}_n$ only appear in $P_1, \ldots, P_n$. Similarly, the type-variables $\Gamma_0\tilde{x}_1, \ldots, \Gamma_0\tilde{x}_n$ do not appear in E. Hence, using the kinded substitution (K', s'), only the new equations and goals need consideration. I show that $s'\Gamma_0 a = s'[l_1 \colon \Gamma_0\tilde{x}_1, \ldots, l_n \colon \Gamma_0\tilde{x}_n]$, and $K' \vdash_{[]\langle\rangle} P_i \blacktriangleright s'\Gamma_0$, for $1 \leq i \leq n$.

By hypothesis, $K' \vdash_{[]\langle\rangle} a \rhd [l_1 : (\tilde{x}_1)\, P_1 \,\&\cdots\&\, l_n : (\tilde{x}_n)\, P_n] \blacktriangleright s\Gamma_0$. By the OBJ$_k$-rule there are $\tilde{\alpha}_i, \Gamma_i$, for $1 \leq i \leq n$, such that

$$K' \vdash_{[]\langle\rangle} P_i \blacktriangleright \Gamma_i \cdot \tilde{x}_i \colon \tilde{\alpha}_i, \tag{1}$$

$$\{a \colon [l_1 \colon \tilde{\alpha}_1, \ldots, l_n \colon \tilde{\alpha}_n]\} \cup \Gamma_1 \cup \cdots \cup \Gamma_n = s\Gamma_0. \tag{2}$$

For the first result, from (2) and the definition of s', one has $s'\Gamma_0 a = s\Gamma_0 a = [l_1 \colon \tilde{\alpha}_1, \ldots, l_n \colon \tilde{\alpha}_n] = s'[l_1 \colon \Gamma_0\tilde{x}_1, \ldots, l_n \colon \Gamma_0\tilde{x}_n]$. For the second result, from (1) and the typing lemma 3.13, for each $1 \leq i \leq n$, one has

$$K' \vdash_{[]\langle\rangle} P_i \blacktriangleright (\Gamma_i \cdot \tilde{x}_i \colon \tilde{\alpha}_i) \restriction \mathrm{fn}\, P_i.$$

Use the WEAK-rule as many times as needed to obtain

$$K' \vdash_{[]\langle\rangle} P_i \blacktriangleright (\Gamma_i \restriction \mathrm{fn}\, P_i) \cup \{\tilde{x}_i \colon \tilde{\alpha}_i\}.$$

From (2) and the definition of s', one has

$$K' \vdash_{[]\langle\rangle} P_i \blacktriangleright s'(\Gamma_0 \restriction \mathrm{fn}\, P_i) \cup \{\tilde{x}_i \colon s'\Gamma_0\tilde{x}_i\}.$$

Finally, use the WEAK-rule as many times as needed to obtain the result. □

I am now in a position to prove the pending results.

PROOF (LEMMA 5.24). (1) Let (K, s) be the most general unifier of (K, E). If (K'', s') is also a unifier of (K, E), there is an s'' such that $s' = s''s$ and (K'', s'') respects K'. Suppose (K'', Δ) is also a well-kinded typing for P. By the completeness invariant at termination, $\Delta \restriction \mathrm{fn}\, P_0 = s'\Gamma_0 \restriction \mathrm{fn}\, P_0 = ss''\Gamma_0 \restriction \mathrm{fn}\, P_0$. Hence $ss''\Gamma_0 \restriction \mathrm{fn}\, P_0 \subseteq \Delta$, and $(K', s\Gamma \restriction \mathrm{fn}\, P)$ is principal for P.
(2) Directly from the completeness invariant at termination. □

PROOF (EXISTENCE AND COMPUTABILITY OF PRINCIPAL TYPINGS, THEOREM 5.19). If P is typable, by the completeness invariant at termination (K, E) is unifiable, and by lemma 5.24 there is a principal kinded-typing for P. □

PROOF (DECIDABILITY OF TYPING ASSIGNMENT IN $\mathrm{TA}_{[]}$, THEOREM 3.24). If P is typable in $\mathrm{TA}_{[]\langle\rangle}$, by theorem 5.19, there is a principal kinded-typing for P. Let it be (K, Δ). Since, by lemma 5.14, subsumption is decidable, check whether (K, Δ) subsumes $(\emptyset, \Gamma)$. If yes, by the subsumption lemma 5.15, $\vdash_{[]\langle\rangle} P \blacktriangleright \Gamma$, and by theorem 5.8(1), $\vdash_{[]} P \blacktriangleright \Gamma$. If not, $\nvdash_{[]\langle\rangle} P \blacktriangleright \Gamma$, and by theorem 5.8(1), $\nvdash_{[]} P \blacktriangleright \Gamma$.

On the other hand, if P is not typable in $\mathrm{TA}_{[]\langle\rangle}$, by theorem 5.8(1), P is not typable in $\mathrm{TA}_{[]}$. □

PROOF (COMPUTABILITY OF TYPING INFERENCE IN $\mathrm{TA}_{[]}$, THEOREM 3.25). Suppose the type inference algorithm produces (Γ_0, K, E). If the unification algorithm 5.21 produces (K', s), by the soundness invariant at termination, $K' \vdash_{[]\langle\rangle} P \blacktriangleright s\Gamma_0$, and by theorem 5.8(2), there is an s' such that $\vdash_{[]} P \blacktriangleright s's\Gamma_0$.

On the other hand, if the unification algorithm fails, by the completeness invariant at termination, P is not typable in $\mathrm{TA}_{[]\langle\rangle}$, and by theorem 5.8(1), P is not typable in $\mathrm{TA}_{[]}$. □

Proof (Decidability of typing assignment in $\mathrm{TA}_{[]\mu}$, theorem 4.17). Replace in the above proof '$\vdash_{[]}$' by '$\vdash_{[]\mu}$', and '$\vdash_{[]\langle\rangle}$' by '$\vdash_{[]\mu\langle\rangle}$'. Use the results 5.12 and 5.16 instead of 5.8 and 5.15. □

Proof (Computability of typing inference in $\mathrm{TA}_{[]\mu}$, theorem 4.18). Replace in the above proof '$\vdash_{[]}$' by '$\vdash_{[]\mu}$', and '$\vdash_{[]\langle\rangle}$' by '$\vdash_{[]\mu\langle\rangle}$'. Use the unification algorithm 6.16 instead of 5.21, and theorem 5.12 instead of 5.8. □

7. Complexity of the Algorithm

Without giving formal definitions for the size of terms, typings, kindings, or sets of equations, one can show that the algorithm 5.23 builds its result in a time and space linear on the size of the input term. For the purposes of this section, it suffices to assume that the size of a term is the number of names and labels the term contains; that the size of a type or a kind is the number of type-variables and labels the type or the kind contains; and that the size of a typing, a kinding, or a set of equations is the sum of the sizes of the types and the kinds they contain. A crucial assumption is that one can access a typing in a constant time.

Theorem 5.30. *Given a term P_0, the algorithm 5.23 builds its result in a time and space linear on the size of P_0.*

Proof. By following the derivation tree of P_0, the initialization step can build Γ_0 in a time proportional to the size of P_0, using a space proportional to the number of distinct names in P_0. This operation, together with the renaming of bound names of P_0 can be done at parsing time.

Now, for the cases in the loop. When the selected goal is a message $a \lhd l : \tilde{v}$, the generated equation and kind assignment take a space proportional to the number of names $a, \tilde{v}$ and label l, and can be computed in a time proportional to the number of names in $a, \tilde{v}$. When the selected goal is an object $a \rhd [l_1 : (\tilde{x}_1)\, P_1 \,\&\, \cdots \,\&\, l_n : (\tilde{x}_n)\, P_n]$, the generated equation takes a space proportional to the number of names $a\tilde{x}_1, \ldots \tilde{x}_n$ and labels $l_1, \ldots l_n$, and take a time proportional to the number of names in $a\tilde{x}_1, \dots \tilde{x}_n$. The result follows by induction. The three remaining cases follow directly by induction. $\square$

It is then clear that the complexity of the type inference algorithm depends on the complexity of the unification algorithm. I shall present in chapter 6 an algorithm to unify kinded sets of equations that runs, under certain conditions, linearly on the size of the kinded set of equations.

8. Notes

Typing subsumption is the distinctive notion shared by systems $\mathrm{TA}_{[]\langle\rangle}$ and $\mathrm{TA}_{[]\mu\langle\rangle}$, and absent in systems $\mathrm{TA}_{[]}$ and $\mathrm{TA}_{[]\mu}$, that provides for the existence of principal typings (cf. [**41**]).

Constraints on the types that may substitute type-variables, in the form of kinds, is due to Ohori [**43**] (also [**13**]), improving a notion of conditional type-schemes by Ohori and Buneman [**44, 45**]. The notions of kindings, kind assignment to types, kinded subsumption, kinded set of equations and unifiers, and the kinded unification algorithm in theorem 5.21 are also due to Ohori.

I would like to stress that kinds are not the only possible solution to the problem of typing subsumption in systems $\mathrm{TA}_{[]}$ and $\mathrm{TA}_{[]\mu}$. One could use, for example, Rémy's fixed-size records with absence and presence information on fields [**48**]. This has the advantage of only needing ordinary unification (as opposed to the more complex kinded unification) but restrains the set of labels to be finite. Both Wand [**71**] and Rémy [**49**] later extended this system to infinite sets of labels. Another possibility is the use of row-variables introduced by Wand [**68, 70**], and refined by Jategaonkar and Mitchell [**34**], and also Stansifer [**55**]. Although the application of these alternatives to the calculus of objects needs deeper investigation, assigning kinds, as a form of open records, to objects referenced (but not defined) in programs seems quite intuitive.

The proof of the equivalence between systems $\mathrm{TA}_{[]}$ and $\mathrm{TA}_{[]\langle\rangle}$ already appear in the first publication of the calculus of objects [**66**]; that of the equivalence of $\mathrm{TA}_{[]\mu}$ and $\mathrm{TA}_{[]\mu\langle\rangle}$ appears in references [**59, 62**]. The typing inference algorithm is based on the one by Honda and myself for the polyadic π-calculus [**65**] (improving an earlier version [**64**]), which in turn is based on Wand's algorithm for the λ-calculus [**69**]. Gay presented an algorithm to extract the most general sorting for a π-term, $\mathcal{O}(p^4)$ on the size p of the input [**21**]. The idea of parsing a term, annotating type (and kind) constraints, postponing all unifications, and performing one single unification at the end, yields a remarkably simple and efficient typing inference algorithm.

The original proofs of the completeness of the typing inference algorithm for the polyadic π-calculus [**65**] and for the calculus of objects [**66**] are not completely correct. In the loop-cases that generate equations (emitters and receptors in the π-calculus; messages and objects in the calculus of objects) the substitution that verifies the invariant *before* and *after* the step cannot be the same, for this does not allow to prove that the substitution solves the equations generated.

CHAPTER 6

Unification of Kinded Infinite Trees

This chapter deals with unification of kinded infinite trees. In particular, I show that if a kinded set of equations is unifiable, it has a most general unifier using only a subset of the variables in the set of equations and in the kinding. I also present an algorithm for extracting the most general unifier of a regular and unifiable kinded set of equations.

To make the chapter independent from the previous ones, some definitions (that of infinite trees, for example) are restated, others are recast into the context of infinite trees. For example, records are now just a particular case of trees, and kindings are maps from variables into infinite trees. The notion of kind assignment to types is replaced by the notion of tree subsumption (not to be confused with kinded-typing subsumption in chapter 5).

Chapter 4 considered trees over an alphabet $\mathbf{F} \cup \mathbf{V}$, where $\mathbf{F}$ is the set of strings of labels in $\mathbf{L}$, and $\mathbf{V}$ is the set of type-variables. Trees in this chapter are defined over arbitrary alphabets F and sets of variables V; the set of record symbols $\mathbf{F}$ (denoted by R in this chapter) is simply a subset of F. In this way, the results in this chapter can be used for different applications, including unification in richer type systems (type systems comprising, for example, a function-space constructor [**13, 43**], or other fixed-size constructors [**56**]) and resolution in logic programming [**54**].

Notation will also change. For example, variables are denoted by $v, v', \dots$ in contrast with $t, t', \dots$ in previous chapters; trees are denoted by $t, u, v, \dots$ in contrast with $\rho, \rho', \dots$; and substitutions on trees are denoted by $\tau, \sigma, \dots$ in contrast with $s, s', \dots$. This agrees with the notation of Courcelle for infinite trees [**18**].

1. Kinded Infinite Trees

DEFINITION 6.1 (Infinite Trees). Assume F to be a ranked alphabet, and V a set of variables, such that $F \cap V = \emptyset$. Variables in V are considered symbols of rank 0.

A *tree over a ranked alphabet* $F \cup V$ is a partial mapping, $u : \mathcal{N}_+^* \to F \cup V$, from the set of strings over positive integers $\mathcal{N}_+$ into symbols in $F \cup V$, satisfying a tree domain condition:

1. The domain of u is prefix closed (that is, if $\tilde{\imath}, \tilde{\jmath} \in \mathcal{N}_+^*$ and $\tilde{\imath}\tilde{\jmath} \in \operatorname{dom} u$, then $\tilde{\imath} \in \operatorname{dom} u$) and non empty;
2. All the siblings at the left of a node in u are nodes in u (that is, if $\tilde{\imath} \in \mathcal{N}_+^*$, $j, k \in \mathcal{N}_+$, $j \leq k$, and $\tilde{\imath}k \in \operatorname{dom} u$, then $\tilde{\imath}j \in \operatorname{dom} u$); and
3. A node of rank k has exactly k subtrees, numbered 1 to k (that is, if $u(i) = f$ of rank k, then for $j \in \mathcal{N}_+$, $\tilde{\imath}j \in \operatorname{dom} u$ if and only if $1 \leq j \leq k$).

Let $M^\infty(F \cup V)$ denote the set of all trees over $F \cup V$, also denoted by $M^\infty(F, V)$. To emphasize the cardinality of V, take $V_n = \{v_1, v_2, \ldots, v_n\}$. Alternative sets of variables are X and W.

Write ε for the empty string of positive integers, and write $\operatorname{root} u$ for $u(\varepsilon)$, the label of the root of tree u. I shall consistently use letter v for variable trees (that is, trees v such that $\operatorname{root} v \in V$), letter t for non-variable trees (that is, when $\operatorname{root} t \in F$), and letter u for either. Sequences of trees $u_1, \ldots, u_n$, for $n \geq 0$, are denoted by $\vec{u}$.

If $u_1, \ldots, u_n$ are n trees, for $n \geq 1$, and the rank of the symbol $f \in F$ is n, write $f(u_1, \ldots, u_n)$ for the tree u such that

$$\begin{aligned} u(\varepsilon) &= f, \\ u(i\tilde{\jmath}) &= u_i(\tilde{\jmath}) \quad \text{if } 1 \leq i \leq n \text{ and } \tilde{\jmath} \in \mathcal{N}_+^*, \\ u(\tilde{\jmath}) &\ \text{is undefined otherwise.} \end{aligned}$$

Now assume L to be a totally ordered ranked alphabet, and build the ranked alphabet R of *record symbols* as the set of strings of pairwise distinct labels in L, such that $l_1 \cdots l_n \in R$ only if $l_1 < \cdots < l_n$. Define the rank of a string $l_1 \cdots l_n$ as $\sum_{i=1}^n \operatorname{rank}(l_i)$, for $n \geq 1$, and the rank of the empty string as 0. Henceforth, assume $R \subseteq F$.

DEFINITION 6.2 (Records). *Records* are finite partial maps $r : L \to M^\infty(F,V)^*$, from the set of labels into strings of infinite trees. The record associated with a (non-variable) tree t in $M^\infty(F,V)$, notation $[t]$, is the empty map if $\operatorname{root} t \in F - R$, and is the map

$$\{l_i \mapsto u_{1+\sum_{j=1}^{i-1}\operatorname{rank} l_j} \dots u_{\sum_{j=1}^{i}\operatorname{rank} l_j} \mid i = 1, \dots, n\}$$

if $\operatorname{root} t \in R$.

Notice that all non variable-trees have an associated record. Non record-trees, that is trees whose root is in $F - R$, are assigned the empty record. In this way, even trees defining, for example, functional types have an associated record: the empty record.

EXAMPLE 6.3. If $\mathit{read}, \mathit{write} \in L$, $\mathit{read} < \mathit{write}$, and $\operatorname{rank} \mathit{read} = \operatorname{rank} \mathit{write} = 1$, then

$$[\mathit{read} \cdot \mathit{write}(\mathit{val}(v), v)] = \{\mathit{read} \mapsto \mathit{val}(v), \mathit{write} \mapsto v\}$$

If $\mathit{cons}, \mathit{nil} \in L$, $\mathit{cons} < \mathit{nil}$, and $\operatorname{rank} \mathit{cons} = 2, \operatorname{rank} \mathit{nil} = 0$, then

$$[\mathit{cons} \cdot \mathit{nil}(v, t)] = \{\mathit{cons} \mapsto vt, \mathit{nil} \mapsto \varepsilon\}$$

If '$\to$' is not a label in R, then $[`\to'(t, t')] = \emptyset$.

DEFINITION 6.4 (Kindings). *Kindings* are total maps $K : V \to M^\infty(F,V)$, such that $\operatorname{root} Kv \in R$ for all $v \in V$. Let t and u be two trees in $M^\infty(F,V)$. Say *t subsumes u under K*, notation $t \succ_K u$, if

$$[t] \subseteq \begin{cases} [Ku] & \text{if } \operatorname{root} u \in V, \\ [u] & \text{otherwise,} \end{cases}$$

so that t subsumes u if the record associated with t has *at most* the components of the record associated with u, or with Ku if u is a variable tree. This is essentially the notion of kind assignment to types in definition 5.3.

EXAMPLE 6.5. For any kinding K, one has $\mathit{read}(\mathit{val}(v)) \succ_K \mathit{read} \cdot \mathit{write}(\mathit{val}(v), v)$, because $[\mathit{read}(\mathit{val}(v))] = \{\mathit{read} \mapsto \mathit{val}(v)\} \subseteq \{\mathit{read} \mapsto \mathit{val}(v), \mathit{write} \mapsto v\} = [\mathit{read} \cdot \mathit{write}(\mathit{val}(v), v)]$.

Let K be a kinding such that $Kv' = \mathit{cons} \cdot \mathit{nil}(v,t)$. Then $\mathit{cons}(v,t) \succ_K v'$, and also $\mathit{nil} \succ_K v'$.

For any kinding K, and trees t, t', t'', one has '$\to$'$(t', t'') \succ_K t$ because ['$\to$'$(t', t'')] = \emptyset$.

2. The Main Result

A (first order) *substitution* is a total map $\tau : V \to M^\infty(G)$, for G a ranked alphabet. Given a substitution τ and a tree u, the tree τu is the result of the simultaneous substitution of tree τv for each occurrence of a variable v in u.

DEFINITION 6.6 (Kinded substitution, unifier). Let $\tau : V \to M^\infty(G, W)$ be a substitution, $K : W \to M^\infty(G, W)$ and $K' : V \to M^\infty(F, V)$ two kindings, and E a set of equations (pairs of trees) in $M^\infty(F, V)$. Pairs (u, u') in a set of equations are written $u \doteq u'$.

1. A *kinded substitution* is a pair (K, τ). Say (K, τ) *respects* K' if $\tau K'v \succ_K \tau v$ for all $v \in V$.
2. *A kinded set of equations* is a pair (K, E). Say (K, τ) is a *unifier* of (K', E) if it respects K', and $\tau u = \tau u'$ for all $u \doteq u' \in E$.
3. Let $(K' : X \to M^\infty(F, X), \tau' : V \to M^\infty(F, X))$ be a kinded substitution. Say (K, τ) is *more general than* (K', τ') if there is a substitution τ'' such that $\tau' = \tau''\tau$, and (K', τ'') respects K.
4. Denote by $\mathrm{Unif}(K, E)$ the set of all unifiers of (K, E), and say (K, E) is *unifiable* if $\mathrm{Unif}(K, E) \neq \emptyset$.

PROPOSITION 6.7. *The most general unifiers of a kinded set of equations are unique up to a renaming of variables.*

PROOF. Suppose that the two kinded substitutions $(K : W \to M^\infty(G, W), \tau : V \to M^\infty(G, W))$ and $(K' : X \to M^\infty(F, X), \tau' : V \to M^\infty(F, X))$ are most general unifiers. There exists a $\theta : W \to M^\infty(F, X)$ such that $\tau' = \theta\tau$ and (K', θ) respects K. Similarly, there exists a $\theta' : X \to M^\infty(F, W)$ such that $\tau = \theta'\tau'$ and (K, θ') respects K'. It is not difficult to see that $\theta'\theta$ is the identity in W, and that X and W are in bijection by θ (see Courcelle [**18**] for details). As for the kindings, since (K, θ') respects K', one has $[\theta' K' v'] \subseteq [K\theta' v']$ for all $v' \in X$. Taking $v = \theta' v'$

(and thus $v' = \theta v$), one successively has: $[\theta' K' \theta v] \subseteq [Kv]$, $\theta[\theta' K' \theta v] \subseteq \theta[Kv]$, and $[K'\theta v] \subseteq [\theta K v]$. But, since (K', θ) respects K, one has $[\theta K v] \subseteq [K'\theta v]$ for all $v \in W$. Hence K' and K are in bijection by θ.

$\square$

Regular trees are trees having finitely many different subtrees. A kinded set of equations (K, E) is regular if the trees in the equations in E, as well as those in the codomain of K, are regular.

I am now a in position to state the main result of this chapter.

THEOREM 6.8. *Let E be a set of equations in $M^\infty(F, V_n)$, and K a kinding $V_n \to M^\infty(F, V_n)$.*

1. *If (K, E) is unifiable, it has a most general unifier (K', τ) with $K' : W_m \to M^\infty(F, W_m)$, $\tau : V_n \to M^\infty(F, W_m)$, and $m \leq n$.*
2. *If (K, E) is regular and unifiable, its most general unifier is regular. It can be effectively computed.*

The machinery needed to prove the theorem is split into two parts. The first concerns *unifiability* of kinded sets of equations: it allows proving the first part of the theorem, and will be dealt in the next section. The second part concerns the *unification algorithm* and its correctness: it allows proving the second part of the theorem, and will be the theme of section 4. The proof of the theorem itself will only appear at the end of section 5.

3. Unifiability

For the rest of this section, let K be a kinding $V_n \to M^\infty(F, V_n)$, and E a set of equations in $M^\infty(F, V_n) \times M^\infty(F, V_n)$. Let $\mathcal{T}$ be the set of all subtrees in the equations in E and in the codomain of K.

DEFINITION 6.9 (Simplifiable relation). An equivalence relation $\sim$ on $\mathcal{T}$ is *simplifiable over a kinding K* if

1. $f(\vec{u}) \sim f(\vec{u}') \;\Rightarrow\; \vec{u} \sim \vec{u}$, and
2. $v \sim v' \;\Rightarrow\; [Kv]l \sim [Kv']l$, for all $l \in \mathrm{dom}[Kv] \cap \mathrm{dom}[Kv']$, and
3. $v \sim t \;\Rightarrow\; [Kv]l \sim [t]l$, for all $l \in \mathrm{dom}[Kv] \cap \mathrm{dom}[t]$,

where $\vec{u} \sim \vec{u}'$ means $u_1 \sim u'_1, \ldots, u_n \sim u'_n$ if $\vec{u} = u_1, \ldots, u_n$ and $\vec{u}' = u'_1, \ldots, u'_n$, for some $n \geq 1$. Denote by $\sim_{KE}$ the least simplifiable equivalence relation over K on $\mathcal{T}$ containing E.

Proposition 6.10. $\mathrm{Unif}(K, E) = \mathrm{Unif}(K, \sim_{KE})$.

Proof. That $\mathrm{Unif}(K, E) \subseteq \mathrm{Unif}(K, \sim_{KE})$ follows by the definition of $\sim_{KE}$. The converse follows by a simple induction of the construction of $\sim_{KE}$. □

Definition 6.11 (Coherent relation). An equivalence relation $\sim$ on $\mathcal{T}$ is *coherent over a kinding* K if one does not have

1. $f(\vec{u}) \sim g(\vec{u}')$ with $f \neq g$, or
2. $v \sim t$ with $\mathrm{dom}[Kv] \not\subseteq \mathrm{dom}[t]$.

I now show how to build the most general unifier (K_p, σ) of a kinded set of equations (K, E). First the substitution σ. Assume that $\sim_{KE}$ is coherent over K, and simply write $\sim$ for $\sim_{KE}$. Let X be the set of variables v in V_n such that

1. $v \sim_{KE} t$, for some non-variable tree t,
2. 1. does not hold, $v = v_i$, and $v_i \sim_{KE} v_j$, for some $j > i$.

Let S be the system of equations $\langle v = u \mid v \in X \rangle$ where u is

1. t if v satisfies 1. above,
2. v_j if v satisfies 2. above, where j is the largest index such that $v \sim_{KE} v_j$.

Let W be $V_n - X$. It is not difficult to prove that S is a generalized Greibach system over $F \cup W$ with set of unknowns X (Courcelle [**18**], lemma 4.9.4). Without loss of generality assume that $X = \{v_1, \ldots, v_m\}$ and that $W = \{v_{m+1}, \ldots, v_n\}$. Let $\langle u_1, \ldots, u_m \rangle$ denote the unique solution of S in $M^\infty(F, W)$, and let $\sigma : V_n \to M^\infty(F, W)$ be the substitution such that

$$\sigma v_i = \begin{cases} u_i & \text{if } v_i \in X, \\ v_i & \text{if } v_i \in W. \end{cases}$$

To build the kinding K_p in the most general unifier of (K, E) I need an auxiliary notion.

DEFINITION 6.12 (Closure of a kinding). A *closure* of $K : V \to M^\infty(F, V)$ over an equivalence relation $\sim$ on $\mathcal{T}$ is a kinding $K^\sim : V \to M^\infty(F, V)$ such that:

$$K^\sim v = l_1 \cdots l_n(\vec{u}_1, \ldots, \vec{u}_n)$$

where $\{l_1, \ldots, l_n\} = \bigcup_{v \sim v'} \mathrm{dom}[Kv']$, $l_1 < \cdots < l_n$, and, for each $1 \le i \le n$, $\vec{u}_i = [Kv']l_i$, for some v' such that $v \sim v'$ and $l_i \in \mathrm{dom}[Kv']$. Notice that if $v \sim v'$ then $K^\sim v = K^\sim v'$, and also $\mathrm{dom}[Kv] \subseteq \mathrm{dom}[K^\sim v']$.

The kinding $K_p : W \to M^\infty(G, W)$ is the restriction of the kinding $\sigma K^{\sim_{KE}}$ to the domain W.

I am now ready to present the main result concerning unifiability.

LEMMA 6.13. 1. *(K, E) is unifiable if and only if $E_K^\sim$ is coherent.*
2. *If (K, E) is unifiable, then (K_p, σ) is a most general unifier of (K, E).*

PROOF. Write $\sim$ for $\sim_{KE}$. By proposition 6.10, one can prove the results with (K, E) replaced by $(K, \sim)$.

i. ($\Rightarrow$) If $\sim$ is not coherent, then $\mathrm{Unif}(K, \sim) = \emptyset$.

($\Leftarrow$) Assume $\sim$ coherent. I show that (K_p, σ) unifies (K, E). Courcelle [**18**] proves that σ unifies E; I need to show that (K_p, σ) respects K, that is, $\sigma K v \succ_{K_p} \sigma v$ for all $v \in V$. I distinguish two cases. If there is a tree t such that $v \sim t$, make $r = [Kv]$ and $r' = [t]$. Since $\sim$ is simplifiable, $rl \sim r'l$, hence $\sigma rl = \sigma r'l$, for all $l \in \mathrm{dom}\, r \cap \mathrm{dom}\, r'$; since $\sim$ is coherent, $\mathrm{dom}\, r \subseteq \mathrm{dom}\, r'$, hence $\mathrm{dom}\, \sigma r \subseteq \mathrm{dom}\, \sigma r'$. Therefore, $[\sigma K v] \subseteq [\sigma t] = [\sigma v]$, and thus $\sigma K v \succ_{K_p} \sigma v$. Otherwise, from the construction of σ one knows that $\sigma v = v_j$ where j is the largest index such that $v \sim v_j$. For each $l \in \mathrm{dom}[K^\sim v_j]$, one knows from the construction of $K^\sim$ that there is a v' such that $v_j \sim v'$ and $[K^\sim v_j]l = [Kv']l$. Also, since $\sim$ is simplifiable, $[Kv]l \sim [Kv']l$, and thus $[\sigma K v]l = [\sigma K v']l = [\sigma K^\sim v_j]l = [K_p \sigma v]l$. Therefore, and since $\mathrm{dom}[Kv] \subseteq \mathrm{dom}[K^\sim v_j]$, one has $[\sigma K v] \subseteq [K_p \sigma v]$, and thus $\sigma K v \succ_{K_p} \sigma v$.

ii. Suppose that (K', τ) also unifies $(K, \sim)$; I show that (K_p, σ) is more general than (K', τ). Courcelle proves that there is a substitution τ' such that $\tau = \tau' \sigma$; I have to show that (K', τ') respects K_p, that is, $\tau' K_p v \succ_{K'} \tau' v$ for all $v \in W$. Since (K', τ) unifies $(K, \sim)$, one has $\tau K v \succ_{K'} \tau v$ for all $v \in V$. Hence $\tau K^\sim v \succ_{K'} \tau v$

for all $v \in W$. But $\tau K^{\sim} v = \tau' \sigma K^{\sim} v = \tau' K_p v$ and $\tau v = \tau' \sigma v = \tau' v$. Therefore $\tau' K_p v \succ_{K'} \tau' v$ for all $v \in W$. $\square$

4. A Unification Algorithm

In view of lemma 6.13, one has to produce an algorithm $\mathcal{U}$ that returns $E_K^{\sim}$ if $E_K^{\sim}$ is coherent, and that signals failure otherwise. The notion of a canonical representative of a tree plays a fundamental role in the algorithm.

DEFINITION 6.14. The *representative of a tree* u under an equivalence relation $\sim$, notation $\mathrm{rep}_{\sim} u$, is a non-variable tree t if there is a t such that $t \sim u$, or a variable tree v such that $v \sim u$ otherwise.

NOTATION 6.15. 1. Denote by $A^{\sharp}$ the symmetric and transitive closure of a binary relation A.

2. $E \cdot u \doteq u'$ means $E \cup \{u \doteq u'\}$ provided $u \doteq u' \notin E$; and $\{\vec{u} \doteq \vec{u}'\}$ means the set of equations $\{u_1 \doteq u_1', \ldots, u_n \doteq u_n'\}$ if $\vec{u} = u_1, \ldots, u_n$ and $\vec{u}' = u_1', \ldots, u_n'$.

The algorithm $\mathcal{U}$ is an adaptation to the kinded setting of Huet's algorithm [**33**], by using transformation rules in the style of Gallier and Snyder [**20**].

ALGORITHM 6.16.
Input. A set of equations E and a kinding K.
Initialization. Set E_0 equal to E, and $\sim_0$ equal the empty set.
Loop. Apply the transformation rules described below, until no rule applies.

i. If $u \sim u'$, then

$$E \cdot u \doteq u' \;,\; \sim \quad \longrightarrow \quad E \;,\; \sim .$$

ii. If i. does not hold, $f(\vec{u}) = \mathrm{rep}_{\sim} u$, and $f(\vec{u}') = \mathrm{rep}_{\sim} u'$, then

$$E \cdot u \doteq u' \;,\; \sim \quad \longrightarrow \quad E \cup \{\vec{u} \doteq \vec{u}'\} \;,\; (\sim \cup \{u \doteq u'\})^{\sharp}.$$

iii. If i. does not hold, $r = [K^{\sim} \mathrm{rep}_{\sim} v]$, and $r' = [K^{\sim} \mathrm{rep}_{\sim} v']$, then

$$E \cdot v \doteq v' \;,\; \sim \quad \longrightarrow \quad E \cup \{rl \doteq r'l \mid l \in \mathrm{dom}\, r \cap \mathrm{dom}\, r'\} \;,\; (\sim \cup \{v \doteq v'\})^{\sharp}.$$

iv. If i. does not hold, $r = [K^{\sim}\,\mathrm{rep}_{\sim} v], r' = [\mathrm{rep}_{\sim} u]$, and $\mathrm{dom}\, r \subseteq \mathrm{dom}\, r'$, then

$$E \cdot v \doteq u\ ,\ \sim \quad \longrightarrow \quad E \cup \{rl \doteq r'l \mid l \in \mathrm{dom}\, r\}\ ,\ (\sim \cup \{v \doteq u\})^{\sharp}.$$

Output. SUCCESS($\sim_n$) if $E_n = \emptyset$; FAIL otherwise.

EXAMPLE 6.17. Consider the kinded set of equations generated by the algorithm 5.23 when given the term $\mathrm{Nil}\, x, \mathrm{Cons}\, xyx$ in example 2.5(6).

Let $V = \{v_x, v_y, v_r, v_r', v_1, v_2\}$; let $K : V \to M^{\infty}(F, V)$ be the kinding

$$\{v_1 \mapsto \mathit{nil}, v_2 \mapsto \mathit{cons}(v_y, v_x), v_x \mapsto v_y \mapsto v_r \mapsto v_r' \mapsto \varepsilon\};$$

and let E be the set of equations

$$\{v_r \doteq v_1, v_x \doteq \mathit{val}(v_r), v_r' \doteq v_2, v_x \doteq \mathit{val}(v_r')\}.$$

The execution of the algorithm is traced by means of the table below. The first column contains the equation selected, the second column the equations generated, the third column the representatives of the new equivalence class in the relation $\sim$, and the fourth column the number of the rule used. The only case worth mentioning occurs in the fourth line, where $\mathrm{rep}_{\sim} v_x = \mathit{val}(v_r)$; $[\mathit{val}(v_r)] = \{\mathit{val} \mapsto v_r\}$; and $[\mathit{val}(v_r')] = \{\mathit{val} \mapsto v_r'\}$; hence the new equation $v_r \doteq v_r'$.

$v_r \doteq v_1$		$v_r \sim v_1$	(3)
$v_x \doteq \mathit{val}(v_r)$		$v_x \sim \mathit{val}(v_r)$	(4)
$v_r' \doteq v_2$		$v_r' \sim v_2$	(3)
$v_x \doteq \mathit{val}(v_r')$	$v_r \doteq v_r'$	$v_x \sim \mathit{val}(v_r')$	(2)
$v_r \doteq v_r'$		$v_r \sim v_r'$	(3)

The generated relation $\sim$ contains three equivalence classes

$$\{v_r, v_r', v_1, v_2\},\ \{v_x, \mathit{val}(v_2)\},\ \{v_y\}.$$

The closure of K under $\sim$ is the kinding

$$K^{\sim} = \{v_r \mapsto v_r' \mapsto v_1 \mapsto v_2 \mapsto \mathit{cons} \cdot \mathit{nil}(v_x, v_y),\ v_x \mapsto v_y \mapsto \varepsilon\}.$$

The set of variables W is $\{v_r, v_y\}$. Then, the substitution $\sigma : V \to M^{\infty}(G, W)$ is the map

$$\{v_r \mapsto v_r' \mapsto v_1 \mapsto v_r, v_x \mapsto \mathit{val}(v_r), v_y \mapsto v_y\}.$$

(There were no infinite trees generated in this case.) Finally, the kinding $K_p : W \to M^\infty(G, W) = \sigma K^\sim \restriction W$ is the map

$$\{v_r \mapsto cons \cdot nil(val(v_r), v_y), v_y \mapsto \varepsilon\}.$$

5. Correctness of the Algorithm

LEMMA 6.18 (Termination). *If (K, E) is regular, $\mathcal{U}(K, E)$ terminates.*

PROOF. If (K, E) is regular, $\sim_{KE}$ is finite. Clearly each $\sim$ is in $\sim_{KE}$. Rule i. reduces $|E|$, the number of equations in E; the remaining rules increase $|\sim|$, the size of $\sim$. One can easily prove that each rule strictly increases $r|\sim| - |E|$, where r is the largest rank of the symbols in F. □

LEMMA 6.19 (Invariant). *At each state n of the algorithm, one has*
1. $\sim_n \subseteq (\bigcup_{i=0}^{n-1} E_i)^\sharp \subseteq (E_n \cup \sim_n)^\sharp$, *for $n \geq 1$, and*
2. $(\bigcup_{i=0}^{n} E_i)^\sharp \subseteq \sim_{KE}$, *for $n \geq 0$.*

PROOF. By induction on n.

i. When $n = 1$, suppose $E_0 = E' \cdot u \doteq u'$. Since $u \not\sim_0 u'$, one has $\sim_1 = \{u \doteq u'\}^\sharp \subseteq E_0^\sharp = (E_1 \cup \sim_1)^\sharp$. When $n > 1$, suppose $E_n = E' \cdot u \doteq u'$. By hypothesis $\sim_n \subseteq (\bigcup_{i=0}^{n-1} E_i)^\sharp \subseteq (E_n \cup \sim_n)^\sharp$. If $u \sim_n u'$, one has $\sim_{n+1} = \sim_n \subseteq (\bigcup_{i=0}^{n-1} E_i)^\sharp \subseteq (\bigcup_{i=0}^{n} E_i)^\sharp = (\bigcup_{i=0}^{n-1} E_i \cup E_n)^\sharp \subseteq ((E_n \cup \sim_n) \cup (E_{n+1} \cup \{u \doteq u'\}))^\sharp = (E_{n+1} \cup \sim_{n+1})^\sharp$. Otherwise $\sim_{n+1} = (\sim_n \cup \{u \doteq u'\})^\sharp \subseteq (\bigcup_{i=1}^{n-1} E_i \cup E_n)^\sharp = (\bigcup_{i=0}^{n} E_i)^\sharp = (\bigcup_{i=0}^{n-1} E_i \cup E_n)^\sharp \subseteq (E_n \cup \sim \cup E_n)^\sharp = (E' \cdot u \doteq u' \cup \sim_n)^\sharp \subseteq (E_{n+1} \cup \sim_{n+1})^\sharp$.

ii. When $n = 0$ one has $E_0^\sharp = E^\sharp \subseteq \sim_{KE}$. When $n > 0$ one has four cases depending on the rule applicable. For the first case, $E_{n+1} \subseteq E_n$. For the remaining cases, notice that, since $\sim_{KE}$ is simplifiable, the equations generated are in $\sim_{KE}$. □

Clearly, upon a successful termination one has $\sim_n = (\bigcup_{i=0}^{n} E_i)^\sharp$. At termination one has the following result.

LEMMA 6.20. *If no rule applies at state n ($n \geq 0$), then $\sim_{KE} \subseteq (\bigcup_{i=0}^{n} E_i)^\sharp$.*

PROOF. By induction on the structure of $\sim_{KE}$. Clearly $E = E_0 \subseteq (\bigcup_{i=0}^{n} E_i)^\sharp$. Write $u \sharp_n u'$ when $u \doteq u' \in (\bigcup_{i=0}^{n} E_i)^\sharp$. Now let $u_1 \sim_{KE} u_2$; without loss of generality, suppose that the equation $u_1 \doteq u_2$ is selected at state j $(0 \leq j < n)$. There are three cases according to the definition of simplifiable relation. In all of them, if rule i. applies, one has $u_1 \sim_j u_2$; and by the invariant $u_1 \sharp_{j-1} u_2$.

Otherwise, if $u_1 = f(\vec{u}_1)$, $u_2 = f(\vec{u}_2)$, one has to show that $\vec{u}_1 \sharp_n \vec{u}_2$. Rule ii. applies; let $\mathrm{rep}_{\sim_j} f(\vec{u}_1) = f(\vec{u}'_1)$, and $\mathrm{rep}_{\sim_j} f(\vec{u}_2) = f(\vec{u}'_2)$. Since $f(\vec{u}_1) \sim_j f(\vec{u}'_1)$, by the invariant $f(\vec{u}_1) \sim_{KE} f(\vec{u}'_1)$; since $\sim_{KE}$ is simplifiable, $\vec{u}_1 \sim_{KE} \vec{u}'_1$; and by induction $\vec{u}_1 \sharp_n \vec{u}'_1$. Similarly, $\vec{u}_2 \sharp_n \vec{u}'_2$. But $\vec{u}'_1 \doteq \vec{u}'_2 \in E_{j+1}$. Therefore $\vec{u}_1 \sharp_n \vec{u}_2$.

Now suppose that $u_1 = v$, $u_2 = t$; one has to show that $[Kv]l \sharp_n [t]l$ for all l in $\mathrm{dom}[Kv] \cap \mathrm{dom}[t]$. If $\mathrm{rep}_{\sim_j} v$ is a non-variable tree, apply rule ii. as above. Otherwise rule iv. applies; let $\mathrm{rep}_{\sim_j} v = v'$ and $\mathrm{rep}_{\sim_j} t = t'$. One has $[K^{\sim_j} v']l \doteq [t']l \in E_{j+1}$ for all l in $\mathrm{dom}[K^{\sim_j} v'] \subseteq \mathrm{dom}[t']$. For each of these l one knows from the construction of $K^{\sim_j}$ that there is a v'' such that $v' \sim_j v''$ and $[K^{\sim_j} v']l = [Kv'']l$. Since $v' \sim_j v''$, following the reasoning for rule ii. above, one has $[Kv']l \sharp_n [Kv'']l$. Similarly, $[Kv]l \sharp_n [Kv'']l$ for all l in $\mathrm{dom}[Kv] \cap \mathrm{dom}[Kv'']$, and $[t]l \sharp_n [t']l$ for all l in $\mathrm{dom}[t] = \mathrm{dom}[t']$. Hence $[Kv]l \sharp_n [t]l$ for all l in $\mathrm{dom}[Kv] \cap \mathrm{dom}[t]$.

The case where rule iii. applies is easier. $\square$

Clearly, at termination one has $\sim_{KE} = (\bigcup_{i=0}^{n} E_i)^\sharp$. I may now state and prove the main result concerning the unification algorithm.

LEMMA 6.21. *Let (K, E) be a regular kinded set of equations.*

1. *$\mathcal{U}(K, E)$ succeeds if and only if $E_K^\sim$ is coherent.*
2. *If $\mathcal{U}(K, E)$ succeeds, it produces $E_K^\sim$.*

PROOF. By lemma 6.18, $\mathcal{U}(K, E)$ terminates; let n $(n \geq 0)$ be the last state.

i. $(\Rightarrow)$ Suppose that $\sim_{KE}$ is not coherent. Then, there is a pair of trees u, u' in $\sim_{KE}$ such that *either* $u = f(\vec{u})$, $u' = g(\vec{u}')$, and $f \neq g$, *or* $\mathrm{root}\, u \in V$, $\mathrm{root}\, u' \in F$, and $\mathrm{dom}[Ku] \not\subseteq \mathrm{dom}[u']$. By lemma 6.20, $\sim_{KE} \subseteq (\bigcup_{i=0}^{n} E_i)^\sharp$; without loss of generality suppose that equation $u \doteq u' \in E_j$ for some j $(0 \leq j \leq n)$. Since no rule applies to this equation, it must be in E_n, and hence $\mathcal{U}(K, E)$ does not succeed.

$(\Leftarrow)$ Suppose that $\mathcal{U}(K, E)$ fails. Then E_n is nonempty, and since no rule applies to E_n, this must contain an equation $u \doteq u'$ with $u \not\sim_n u'$ and *either* $\mathrm{rep}_{\sim_n} u = f(\vec{u})$, $\mathrm{rep}_{\sim_n} u' = g(\vec{u}')$, and $f \neq g$, *or* $\mathrm{rep}_{\sim_n} u = v$, $\mathrm{rep}_{\sim_n} u' = t$, and

$\mathrm{dom}[K^{\sim_n} v] \not\subseteq \mathrm{dom}[t]$. In either case $E_n^\sharp$ is not coherent, and by lemma 6.19.ii. $\sim_{KE}$ is not coherent.

ii. If $\mathcal{U}(K, E)$ succeeds, $E_n = \emptyset$. By lemmas 6.19 and 6.20, $\sim_n = \sim_{KE}$. □

I am now in a position to prove the main result of this chapter.

Proof (theorem 6.8). *i.* The first part follows directly from lemma 6.13.ii. The second part follows from the fact that $W \subseteq V_n$.

ii. For the first part, notice that if (K, E) is regular, then $\mathcal{T}$ is a set of regular trees, and S an extended regular system whose solution consists of regular trees, and thus both σ and K_p are regular. For the second part, notice that if (K, E) is unifiable, by lemma 6.13.i, $\sim_{KE}$ is coherent, and, since (K, E) is regular, by lemma 6.21, $\mathcal{U}(K, E)$ succeeds producing $\sim_{KE}$. □

6. Complexity of the Algorithm

By implementing the equivalence relation with the Aho, Hopcroft, and Ullman's union-find method [**3**] (with balanced trees and path compression), and based on the analysis of Tarjan [**57**] and Aho, Hopcroft, and Ullman [**3**], Huet [**33**] concludes that the unification of conventional regular infinite trees is *quasi*-linear on the size of the set of equations. More precisely if n is the number of different subtrees in the input, then the algorithm runs in $O(n \cdot \mathrm{ack}^{-1}(n))$, where ack is the Ackerman function. (Observe that ack^{-1} grows extremely slowly: for example $\mathrm{ack}^{-1}(65536) = 5$.)

When comparing algorithm $\mathcal{U}$ with that of Huet, one notices the following additional operations in rules (3) and (4).

1. Computing the record associated with a tree;
2. Computing $K^\sim$, the closure of kinding K with respect to relation $\sim$;
3. Accessing the components of records, and
4. Computing record domain and testing record inclusion.

Nodes representing records can maintain explicit record information. This avoids computing the record associated with trees whenever these are needed. Item 2 above can be obviated by maintaining $K^\sim$ explicitly. This amounts to initialize $K^\sim$ to K, and to update it in rule (3), merging the components of the two records r and r' (ties broken arbitrarily), and assigning the new record to the representative of variable v. (No such operation is needed in rule (4) since, after the application of the rule, the representative of v is not a variable anymore.)

All that remains is to test record domains for inclusion, computing record domain intersection, and accessing the components of records. Assuming the set of labels $\mathbf{L}$ finite, these operations are known to be executable in constant time, making the time complexity of algorithm $O(n\,\mathrm{ack}^{-1}(n))$. Without such an assumption, testing record domains and accessing their components is logarithmic on the size of the records, slowing the execution of the algorithm by a logarithmic factor.

If $\sim$ is implemented with the union-find method, then the union operation joins two equivalence classes, and the find operation finds the representative of a tree. In this case, care must be taken in rule iv. to join v to u and not vice-versa.

7. Notes

The analysis of unification of kinded infinite trees closely follows that of Courcelle for ordinary infinite trees ([**18**], section 4.9), which in turn is based on that of Huet ([**33**], chapter 5). The notions of simplifiable and coherent relations, and the building of the most general unifier of a simplifiable and coherent relation are due to Huet. I simply adapted them to the kinded setting. A crucial point is the closing of a kinding under an equivalence relation, without which the results on unifiability could not be obtained.

The original "fast-unification" algorithm is said [**8, 52**] to have been discovered independently by Huet [**33**], Robinson [**51**], and Baxter [**9**]. The kinded unification algorithm is an adaptation of Huet's algorithm, using transformation rules in the style of Gallier and Snyder [**20**]. When compared with Huet's algorithm, the present algorithm has one more rule: it distinguishes the unification of two variables from that of a variable and a tree.

The first algorithm to unify kinded sets of equations (over finite terms) is due to Ohori, and comprises six transformation rules. By treating records non differently from ordinary trees, I could collapse the rules for record/record and tree/tree in Ohori's algorithm into a single rule (rule (2)). Also, by assigning the empty record to non-record trees, I could collapse the rules for variable/record and variable/tree into a single rule (rule (3)).

CHAPTER 7

Comparisons, Extensions, and Applications

This chapter compares the contents of this thesis with related work, presents extensions to the basic calculus and its typing system towards a more effective modelling of concurrent object-oriented computations, and describes applications to concrete programming languages.

1. The π-Calculus as a Calculus of Objects

Not surprisingly, the π-calculus, the ν-calculus, and their typing systems can be faithfully encoded in the calculus of objects. Honda has shown how to encode the π-calculus into the ν-calculus [**28**]; all I need to show is how to encode the ν-calculus into the calculus of objects. The ν-calculus arises from the π-calculus by dropping the continuation process of an output-guarded process, that is, by replacing $\overline{a}v.P$ by $\overline{a}v.\mathbf{0}$, written in ν-calculus as $\leftarrow av$. In this section I shall consider the ν-calculus with replication [**30**], and the type system for the π-calculus [**64, 65**] without recursive types. The ν-calculus is essentially monadic; the extension to the polyadic case is however straightforward.

Pick a label, say *val*, and make it the unique label in the set of labels. Messages in the ν-calculus become tagged with *val*. On the other hand, ν-receptors are translated into objects with a single method labelled with *val*. Part of the translation $[\![\cdot]\!]$ from ν-terms into terms in the calculus of objects can be defined as follows (the remaining constructors are shared by both calculi).

$$[\![\leftarrow av]\!] = a \lhd \mathit{val}{:}\, v$$

$$[\![a(x)\,.P]\!] = a \rhd [\mathit{val}{:}\,(x)\,[\![P]\!]]$$

It is then easy to prove that ν-terms and their encodings behave *strongly* similarly. Types in the π-calculus can also be easily encoded in types in $\mathbf{T}_{[]}$. The encoding of a type-variable is the type-variable itself; constructed types become records with a single component labelled with *val*.

$$[\![t]\!] = t$$

$$[\![(\alpha)]\!] = [\mathit{val}\colon [\![\alpha]\!]]$$

Such a translation naturally extends to typings by translating the predicates and maintaining the subjects. Using the type system $\mathrm{TA}_{()}$ for the π-calculus [**65**], a simple induction on the length of deductions shows that

$$\vdash_{()} P \blacktriangleright \Gamma \quad \text{if and only if} \quad \vdash_{[]} [\![P]\!] \blacktriangleright [\![\Gamma]\!].$$

2. The Calculus of Objects Can't Be a Typed π-Calculus

There seems to be no method of encoding *general* terms in the calculus of objects into the *typed* π- (or ν-) calculus. Two particular cases that allow translation—when the set of labels is finite, and when the targets of all messages are also locations of some object—are discussed below. For simplicity I shall consider the monadic version of the calculus of objects.

When the set of labels is finite. To make synchronization simpler I invert the roles of objects and messages: messages become π-receptors, and objects π-emitters. Suppose $\mathbf{L} = \{l_1, \ldots, l_n\}$. The encoding of an object outputs a sequence of n fresh names, each of which representing a label in $\mathbf{L}$, and giving access to a different method. As a consequence, when objects have less than n methods, some labels will give access to no method. The encoding of a message accepts a sequence of n names, selects one of them and invokes the method with this name. A translation $[\![\cdot]\!]$ from terms in the calculus of objects into the polyadic π-calculus can be defined as follows.

$$[\![a \lhd m : v]\!] = a(l_1 \cdots l_n)\,.\overline{m}v.\mathbf{0}$$

$$[\![a \rhd [m_1 : (x)\, P_1 \;\&\; \cdots \;\&\; m_k : (x)\, P_k]]\!] =$$

$$\nu l_1 \cdots l_n(\overline{a}l_1 \cdots l_n.\mathbf{0} \mid m_1(x)\,.[\![P_1]\!] \mid \;\cdots\; \mid m_k(x)\,.[\![P_k]\!])$$

Notice that $\{m, m_1, \ldots, m_k\} \subseteq \mathbf{L} = \{l_1, \ldots, l_n\}$. It should be easy to show that terms in the calculus of objects and their corresponding π-terms behave *weakly* similarly. The encoding is particularly inefficient when the cardinal of $\mathbf{L}$ is large and objects have few methods on average, since for each message passing a sequence of n names must be created and exchanged.

Types in the calculus of objects may now be encoded into π-types. A record-type becomes a tuple of fixed-length n, the components not present in the record being filled with a fixed π-type Empty.

$$\llbracket t \rrbracket = t$$
$$\llbracket [m_1\colon \alpha_1, \ldots, m_k\colon \alpha_k] \rrbracket = (\beta_1 \cdots \beta_n)$$
$$\text{where } \beta_i = \begin{cases} (\llbracket \alpha_i \rrbracket) & \text{if } l_i \in \{m_1, \ldots, m_k\}, \\ \text{Empty} & \text{otherwise.} \end{cases}$$

As before, this translation extends to typings by translating the predicates and keeping the subjects. It is then not difficult to prove by induction on the length of deductions that

$$\vdash_{[]} P \blacktriangleright \Gamma \quad \text{if and only if} \quad \vdash_{()} \llbracket P \rrbracket \blacktriangleright \llbracket \Gamma \rrbracket.$$

When the set of labels is not finite. Things get more complicated when the set of labels is not finite, or when one is not willing to exchange, for each communication, as many names as there are labels. One could try an approach similar to the case of a finite number of labels, and make objects propose, not n distinct fresh names, but only as many names as the object's methods. Then, assuming a total order on labels, such that, for example, $m_1 < \cdots < m_k$, one could encode objects as

$$\llbracket a \triangleright [m_1 : (x)\, P_1 \,\&\, \cdots \,\&\, m_k : (x)\, P_k] \rrbracket \quad = \quad \nu m_1 \ldots m_k (\overline{a} m_1 \ldots m_k.\mathbf{0} \mid m_1(x)\,.\llbracket P_1 \rrbracket \mid \cdots \mid m_k(x)\,.\llbracket P_k \rrbracket)$$

thus replacing the label selection of methods by a *positional* selection.

But what to do with messages? How should one encode a message $a \triangleleft \mathit{write}\colon v$? If one knows that name a is the location of an object capable of receiving, say, three different kinds of message, of which *write* labels the second (in the label ordering), then one may encode message $a \triangleleft \mathit{write}\colon v$ as

$$a(m_1 m_2 m_3)\,.\overline{m}_2 v.\mathbf{0}$$

If, one the other hand, no a-located object is known, it is not possible to determine the offset of label *write*.

There is a simple procedure to determine when the encoding is possible, and in this case to determine the offset of labels. Use the typing inference algorithm 5.23 on the input program to obtain (Γ_0, K, E); use then a unification algorithm to obtain the most general unifier (K', s) of (K, E). If there is a non-empty kind in K', the encoding is not possible. Otherwise, to determine the offset of label l in a message $a \lhd l\!:\!v$, check the record $s\Gamma_0 a$ (notice that $s\Gamma_0 a$ is not a type-variable, otherwise $K'(s\Gamma_0 a)$ would not be the empty kind).

When the encoding of terms is possible, the encoding of types becomes simpler than in the case of a finite number of labels.

$$\llbracket t \rrbracket = t$$
$$\llbracket [m_1 \colon \alpha_1, \ldots, m_k \colon \alpha_k] \rrbracket = ((\llbracket \alpha_1 \rrbracket) \cdots (\llbracket \alpha_k \rrbracket))$$

As in the previous case, it is not difficult to prove that

$$\vdash_{[]} P \blacktriangleright \Gamma \quad \text{if and only if} \quad \vdash_{()} \llbracket P \rrbracket \blacktriangleright \llbracket \Gamma \rrbracket.$$

Pierce, Rémy, and Turner followed a similar approach encoding objects in the π-calculus [**47**]. The programming language they propose enforces the declaration of objects *before* allowing message sends, and, as such, the encoding is always possible. I used a two-way protocol (send-receive, as seen from the object side), while they proposed a three-way protocol (receive-send-receive).

Walker proposed a different approach to encode objects in the π-calculus, by making labels constant-names, and by using sums of matches to encode labelled methods [**67**]. Using his idea, an object $a \rhd [l_1\!:\!P_1 \,\&\, \cdots \,\&\, l_k\!:\!P_k]$ could be translated into the following π-process:

$$a(l)\,.([l = l_1]\llbracket P_1 \rrbracket + \cdots + [l = l_k]\llbracket P_k \rrbracket)$$

The problem is in passing arguments to the methods, while avoiding interferences from external parties and maintaining typability (Walker's solution is not typable in general). For interference-free communications, arguments must be sent either together with l on name a, or via a private channel established between the message and the object. Unless all methods have the same type, the first alternative would not be typable. As for the establishment of a private channel, this must be done *before* method selection, for otherwise the object loses contact with the message. Then one is in the case of the first alternative where the same name (the private channel in this case) must be used to communicate arguments to different methods.

Ohori studied the problem of compiling labelled records into a ML-like polymorphic record-calculus, by encoding records as arrays, labels as indices in arrays, and by passing to functions not only the array representing the record, but also the indices of the labels used in the body of the function [**43**]. Such a method does not work in a name-passing calculus because objects are sent names (or references, as opposed to values), and there may be no object associated with a given name.

3. Polymorphism

Polymorphism refers to the fact that the same program can be used with values of different types. Consider the buffer cell defined in example 2.5(5). To instantiate two cells, say an integer cell and a boolean cell, from the same template one can naturally write

$$\nu x(x \lhd \mathit{recur}\colon \mathrm{intcell}\,1, x \lhd \mathit{recur}\colon \mathrm{boolcell}\,\mathrm{true},$$
$$!x \rhd [\mathit{recur}\colon (sv)\, s \rhd [\mathit{read}\colon (r)\, r \lhd \mathit{val}\colon v, x \lhd \mathit{recur}\colon sv \;\&\; \mathit{write}\colon (u)\, x \lhd \mathit{recur}\colon su]]).$$

Such a program is however not typable, for name x carries a message *recur* with two different types. To account for this kind of polymorphism, I introduced a notion of agents (as processes abstracted on a series of types), and variables over agents (cf. the constants in example 2.5(4)) that allow writing a program once and using it several times in different contexts [**61**]. This allows in particular declaring a generic buffer cell (the *class* of buffer cells), and instantiating several copies of the cell, each buffering a name of a different type.[1]

$$\mathbf{letrec}\ \mathrm{Cell} = (sv)\, s \rhd [\mathit{read}\colon (r)\, r \lhd \mathit{val}\colon v, \mathrm{Cell}(sv) \;\&\; \mathit{write}\colon (u)\, \mathrm{Cell}(su)]$$
$$\mathbf{in}\ \mathrm{Cell}(\mathrm{intcell}\,1), \mathrm{Cell}(\mathrm{boolcell}\,\mathrm{true})$$

[1] As usual, **letrec** $X = A$ **in** P is short for **let** $X =$ **rec** $X.A$ **in** P.

A notion of ML-polymorphism [**19**] or predicative polymorphism (in the sense that polymorphic types are only defined after the monomorphic types have been defined [**41**]) allows assigning Cell a polymorphic type

$$\forall t.[\mathit{read}\colon [\mathit{val}\colon t], \mathit{write}\colon t] \;\; t$$

and instantiating this type into a sequence of simple (monomorphic) types

$$[\mathit{read}\colon [\mathit{val}\colon \mathrm{Int}], \mathit{write}\colon \mathrm{Int}] \;\; \mathrm{Int}$$

to be assigned to names intcell and 1, respectively. Similarly, one can instantiate the polymorphic type for Cell into

$$[\mathit{read}\colon [\mathit{val}\colon \mathrm{Bool}], \mathit{write}\colon \mathrm{Bool}] \;\; \mathrm{Bool}$$

and assign the thus obtained types to names boolcell and true, respectively.

I studied this system in the context of the polyadic π-calculus [**61**], and showed that it preserves all the interesting properties of the monomorphic type system, including subject-reduction and computability of principal typings. Since the new notions are orthogonal to the structure of terms and types, a new layer of predicative polymorphism can be easily built on top of the monomorphic type system developed in this thesis. This form of polymorphism effectively provides for *classes* as objects abstracted on a series of names (the name "self" and the object's local variables), and for objects instances of classes.

A different form of polymorphism was studied by Turner [**58**], also in the setting of the π-calculus. The idea is to assign (impredicative) polymorphic types directly to names, thus obtaining a name-passing counterpart of the second order λ-calculus [**22, 50**]. The system thus obtained is more flexible than the one with predicative polymorphism, allowing in particular typing the first example in this section. Nonetheless, type inference for impredicative systems remains an open problem, making it impractical for implicitly typed programming languages.

4. TyCO: Towards TYped Concurrent Objects

I developed a prototype of an implicitly typed polymorphic concurrent programming language based on the calculus of objects augmented with the notions developed in section 3: agents, variables over agents, and predicative polymorphism [**63**].

An interactive interpreter written in Standard ML accepts either processes or process-declarations. In the former case it finds the principal typing of the process, outputs it, and places the process in parallel with the already existing processes. In the latter case, the interpreter outputs the principal type of the agent and remembers this. The type inference system has been running for almost two years; the interpreter is not yet written at the time I am writing this thesis. A typical session on TyCO is as follows.

```
> rec Cell = (cell val).
>   self :>
>     [ read:  (replyto) replyto <:  val:  val, Cell(cell val)
>     & write:  (newval) Cell(cell newval)];
Cell:  forall t.[read:  <val:  t>, write:  t]
```

```
> Cell(intcell 1);
intcell:  [read:  <val:  Int>, write:  Int]
> Cell(boolcell true);
boolcell:  [read:  <val:  Bool>, write:  Bool]
> intcell <:  write;
Error:  wrong number of arguments in communication.
    Expected:  Int; found:  .
> Cell(somecell);
Error:  wrong number of arguments in agent application.
    Expected:  [read:  <val:  t>, write:  t] t; found:  t'.
> intcell <:  put;
Error:  label put not found
    in record [read:  <val:  t>, write:  t].
```

The calculus of objects is quite low-level: the programmer has to write a large amount of synchronization code otherwise hidden in higher-level programming languages, the λ-calculus included. This low-level of abstraction leads to non-reliable and difficult to maintain programs. A number of high-level programming constructors can be defined on top of the basic calculus, including values (as expressions evaluating to names), case-statements (similar to the conditional statement of example 2.5(3)), and class and subclass declarations. See reference [**63**] for details.

5. Semantics and Type Systems for Concurrent Languages

One of the goals of this thesis is to provide for semantics and type systems for existing (eg. ABCL [**72**], and POOL [**4**]) and future concurrent object-oriented programming languages. To test the calculus of objects and its typing system, I picked the language ABCL/1. Operational semantics for ABCL were given via a systematic translation of the ABCL program constructors into processes; a type system for ABCL source code is derived from this translation and the type system of the calculus of objects; the results can be found in reference [**60**].

Each of the ABCL forms is encoded into a process that, upon evaluation, produces its result on a given name. This name can be thought of as the continuation of the form, and allows sequentializing the various forms that composed ABCL methods. Most of the forms have a simple encoding; in particular most ABCL-objects can be encoded with a single object in the calculus of objects (plus those implementing the local-variables of the object).

There are however two kinds of ABCL constructors that do not allow a simple encoding: they are *wait-for forms* and *express messages*. The former allows objects to dynamically specify the kind of messages they want to receive. Express-messages, on the other hand, can interrupt the execution of methods associated with a regular message. The encoding of these two constructors requires a more complex model where message queues have to be made explicit. An ABCL-object is then encoded as a society of objects with a receptor object accepting and enqueuing messages, one queue object for each kind of message, an executor to execute the methods themselves, express-checkers to check the arrival of express-messages, and servers dequeuing messages from the enabled methods and feeding them to the executor. The complexity of the encoding indeed shows how low-level the calculus of objects is.

A type system for ABCL source programs is obtained from the encoding. It forms a set of admissible rules of the type system of the calculus of objects, thus enabling the ABCL type-system to inherit all the properties of the calculus of objects, including lack of runtime errors and computability of principal typings.

6. Other Calculi

Starting with Cardelli's work [**14**], a lot of attention has been dedicated to the incorporation of labelled-records into the λ-calculus. This approach does not provide for concurrency (although one could think of adding an ad-hoc concurrency operator to the λ-calculus), and stateful objects are difficult to implement. Another kind of difficulty arises when one thinks about types. The name "self" and mutable local-variables for objects often lead to unnecessary recursive type considerations.

The calculus of objects, on the other hand, by treating "self" and local-variables no differently from other names, achieves a remarkable simplicity on types. There is no form of function abstraction and application in the calculus, but this can be encoded into the basic calculus as demonstrated by Milner [**37**] (cf. [**47, 63**]).

Nierstrasz proposed a name-passing calculus featuring communication of tuples and function application as primitives [**42**]. As demonstrated by several works [**37, 47, 63**], function application can be cleanly incorporated into pure name-passing calculi.

Abadi and Cardelli recently proposed a basic calculus supporting builtin objects, and method invocation and override [**1**]. Although no form of concurrency is present, the calculus shares with the calculus of objects the notions of builtin objects composed of labelled methods, and that of labelled method invocation.

CHAPTER 8

Concluding Remarks

Aiming at providing for solid operational semantics and type systems for concurrent object-oriented programming languages, this thesis comprises the proposal of an untyped name-passing calculus, the study of type systems for the calculus, and an investigation of unification of kinded infinite trees.

The calculus of objects is a name-passing calculus built along the lines of Milner's π-calculus, its polyadic form, and Honda's ν-calculus; its distinctive characteristics being labelled sums denoting methods in objects, and labelled asynchronous messages selecting a particular method in the target object.

The type systems assign types to names, and typings (that is, sets of name-type pairs) to terms. Simple types are built from type-variables by means of a single record constructor that uses the labels in the syntax of the calculus. To allow typing terms containing circular name-structures, a recursive type constructor is introduced, recursive types being viewed as regular (that is, having finitely many subtrees) infinite trees. Nonetheless, no notion of typing subsumption, and hence of principal typings, seems to be possible in these systems. I use constraints on the types that may replace a given type-variable, and study the problem of unifying infinite trees with variables subject to constraints.

The accomplishments of this study can be summarized as follows:

1. The proposal of a *name-passing calculus* incorporating labels as a primitive notion, labelled asynchronous messages, and labelled sums as methods in objects, suited to describe concurrent object-oriented computations;
2. the study of decidable type systems that allow typing processes containing recurring name-structures guaranteeing that typable processes do not run into errors;

3. the definition of the conditions under which kinded infinite trees are unifiable, and the description of an algorithm for unifying regular kinded infinite trees.

The relevance of the calculus of objects is attested by the natural encoding of the concurrent object-oriented programming language ABCL/1, thus providing for an operational semantics and a typing system for the language. Predicative polymorphism, applicative (functional-like) behavior, as well as classes, subclasses and inheritance can be added as new layers to the basic calculus and its typing system, towards a more effective modeling of concurrent objects.

Bibliography

[1] Martín Abadi and Luca Cardelli. A theory of primitive objects. DEC-SRC, Digital Equipment Corporation, Systems Research Laboratory, 1994. Shorter versions in European Symposium on Programming, and in Theoretical Aspects of Computer Software.

[2] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

[3] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[4] Pierre America. POOL-T: A parallel object-oriented language. In Akinori Yonezawa and Mario Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 199–220. MIT Press, 1987.

[5] H. P. Barendregt. *The Lambda-Calculus, its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1974. Revised edition.

[6] H. P. Barendregt. Lambda calculi with types. In S. Abramsky, Dov M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, pages 118–309. Oxford University Press, 1992.

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

[8] Jonas Barklund. *Parallel Unification*. PhD thesis, Uppsala University, 1990.

[9] L. D. Baxter. *The Complexity of Unification*. PhD thesis, University of Waterloo, 1976. Cited in [**52**].

[10] Gérard Berry and Gérard Boudol. The chemical abstract machine. In *17th ACM Symposium on Principles of Programming Languages*, pages 81–94. ACM Press, 1990. Also in Theoretical Computer Science, 96:217–248, 1992.

[11] Gérard Boudol. Some chemical abstract machines. INRIA Sophia Antipolis, 1994.

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

[13] Peter Buneman and Atsushi Ohori. Polymorphism and type inference in database programming. Technical report, University of Pennsylvania, 1992. To appear in ACM Transactions on Database Systems.

[14] Luca Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76:138–164, 1988.

[15] 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.

[16] Felice Cardone and Mario Coppo. Type inference with recursive types: Syntax and semantics. *Information and Computation*, 92:48–80, 1991.

[17] B. Courcelle, G. Kahn, and J. Vuillemin. Algorithmes d'equivalence et de reduction a des expressions minimales dans une classe d'equations recursives simples. In *2nd Automata, Languages and Programming*, volume 14 of *LNCS*, pages 200–214. Springer-Verlag, 1974.

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

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

[20] Jean H. Gallier and Wayne Snyder. Complete sets of transformations for general E-unification. *Theoretical Computer Science*, 67:203–260, 1989.

[21] Simon J. Gay. A sort inference algorithm for the polyadic π-calculus. In *20th ACM Symposium on Principles of Programming Languages*, pages 429–438. ACM Press, 1993.

[22] Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérior*. Thèse d'état, Université de Paris VII, 1972. Cited in [**41**].

[23] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[24] Carl Hewitt. Control structures as patterns of passing messages. *Artificial Intelligence*, 8(3):323–364, 1977.

[25] 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.

[26] Roger Hindley and Jonathan Seldin. *Introduction to Combinators and λ-Calculus*. Cambridge University Press, 1986.

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

[28] 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.

[29] Kohei Honda and Mario Tokoro. On asynchronous communication semantics. In *Object-Based Concurrent Computing*, volume 612 of *LNCS*, pages 21–51. Springer-Verlag, 1992.

[30] 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.

[31] Kohei Honda and Nobuko Yoshida. Combinatory representation of mobile processes. In *21st ACM Symposium on Principles of Programming Languages*, pages 348–360. ACM Press, 1994.

[32] Kohei Honda and Nobuko Yoshida. Replication in concurrent combinators. In *2nd Theoretical Aspects of Computer Software*, volume 789 of *LNCS*, pages 786–805. Springer-Verlag, 1994.

[33] Gérard Huet. *Résolution d'équations dans des langages d'ordre* $1, 2, \ldots, \omega$. Thèse d'État, Université de Paris VII, 1976.

[34] Lalita A. Jategaonkar and John C. Mitchell. ML with extend pattern matching. In *ACM Conference on LISP and Functional Programming*, pages 198–211. ACM Press, 1988.

[35] David MacQueen, Gordon Plotkin, and Ravi Sethi. An ideal model for recursive polymorphic types. *Information and Control*, 71:95–130, 1986.

[36] Robin Milner. *Communication and Concurrency.* C. A. R. Hoare Series Editor. Prentice-Hall Int., 1989.

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

[38] Robin Milner. Sorts and types in the π-calculus. University of Edinburgh, December 1990.

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

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

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

[42] Oscar Nierstrasz. Towards an object calculus. In *Object-Based Concurrent Computing*, volume 469 of *LNCS*, pages 1–20. Springer-Verlag, 1992.

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

[44] Atsushi Ohori and Peter Buneman. Type inference in a database programming language. In *ACM Conference on LISP and Functional Programming*, pages 174–183. ACM Press, 1988.

[45] Atsushi Ohori and Peter Buneman. Static type inference for parametric classes. In *4th Object Oriented Programming: Systems, Languages, and Applications*, pages 445–455, 1989. Also in

Theoretical Aspects of Object-Oriented Programming, chapter 5, pages 121–147. MIT Press, 1993.

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

[47] Benjamin C. Pierce, Didier Rémy, and David N. Turner. A typed higher-order programming language based on the pi-calculus. University of Edinburgh, 1993.

[48] Didier Rémy. Typechecking records and variants in a natural extension of ML. In *16th ACM Symposium on Principles of Programming Languages*, pages 77–87. ACM Press, 1989.

[49] Didier Rémy. Type inference for records in a natural extension of ML. In *Theoretical Aspects of Object-Oriented Programming*, chapter 3, pages 121–147. MIT Press, 1993.

[50] J. C. Reynolds. Towards a theory of type structure. In *Paris Symposium on Programming*, volume 19 of *LNCS*, pages 408–425, 1974.

[51] John A. Robinson. Fast unification. In *Tagung über Automatisches Beweisen*. Mathematisches Forschungsinstitut Oberwolfach, 1976. Cited in [**52**].

[52] John A. Robinson. Logic and logic programming. *Communications of the ACM*, 35(3):40–65, March 1992.

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

[54] Gert Smolka and Ralf Treinen. Records for logic programming. In Krzysztof Apt, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 240–254. MIT Press, November 1992. Also in RR–92–23, Deutsches Forschungszentrum für Künstliche Intelligenz, 1992.

[55] Ryan Stansifer. Type inference with subtypes. In *15th ACM Symposium on Principles of Programming Languages*, pages 88–97. ACM Press, 1988.

[56] Kaku Takeuchi, Kohei Honda, and Makoto Kubo. An interaction-based programming language and its typing system. In *5th Parallel Architectures and Languages Europe*, LNCS. Springer-Verlag, July 1994.

[57] R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22:215–225, 1975.

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

[59] Vasco T. Vasconcelos. Recursive types in a calculus of objects. CS 93–002, Keio University, November 1993. Revised version in [**62**].

[60] 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.

[61] Vasco T. Vasconcelos. Predicative polymorphism in π-calculus. In *6th Parallel Architectures and Languages Europe*, volume 817 of *LNCS*, pages 425–437. Springer-Verlag, July 1994.

[62] Vasco T. Vasconcelos. Recursive types in a calculus of objects. *Transactions of Information Processing Society of Japan*, 35(9):1828–1836, September 1994.

[63] Vasco T. Vasconcelos. Typed concurrent objects. In *8th European Conference on Object-Oriented Programming*, volume 821 of *LNCS*, pages 100–117. Springer-Verlag, July 1994.

[64] Vasco T. Vasconcelos and Kohei Honda. Principal typing-schemes in a polyadic π-calculus. CS 92–004, Keio University, November 1992. Abridged version in [**65**].

[65] 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 92–004.

[66] 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.

[67] David Walker. π-calculus semantics of object-oriented programming languages. In *Theoretical Aspects of Computer Software*, volume 526 of *LNCS*, pages 532–547. Springer-Verlag, 1992.

[68] Mitchell Wand. Complete type inference for simple objects. In *2nd IEEE Symposium on Logic in Computer Science*, pages 37–44, June 1987.

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

[70] Mitchell Wand. Corrigendum: complete type inference for simple objects. In *3rd IEEE Symposium on Logic in Computer Science*, page 132, June 1988.

[71] Mitchell Wand. Type inference for record concatenation and multiple inheritance. In *4th IEEE Symposium on Logic in Computer Science*, pages 92–97, June 1989.

[72] Akinori Yonezawa, editor. *ABCL, an Object-Oriented Concurrent System*. MIT Press, 1990.

[73] Akinori Yonezawa and Mario Tokoro, editors. *Object-Oriented Concurrent Programming*. MIT Press, 1987.

Index