INSTITUTO SUPERIOR TÉCNICO

UNIVERSIDADE TÉCNICA DE LISBOA

INSTITUTO SUPERIOR TÉCNICO

Typing Non-uniform Concurrent Objects

António Maria Alarcão Ravara

(Mestre)

Dissertação para Obtenção do Grau de Doutor em Matemática

Orientador:	Doutor Amílcar dos Santos Costa Sernadas
Co-orientador:	Doutor Vasco Manuel Thudichum de Serpa Vasconcelos

Júri

Presidente:	Reitor da Universidade Técnica de Lisboa
Vogais:	Doutor Amílcar dos Santos Costa Sernadas
	Doutor José Félix Gomes da Costa
	Doutor Kohei Honda
	Doutor Vasco Manuel Thudichum de Serpa Vasconcelos
	Doutor Pedro Manuel Agostinho Resende
	Doutor Gérard Boudol

Agosto de 2000

Tipificação de Objectos Concorrentes Não Uniformes

Nome: António Maria Alarcão Ravara

Doutoramento em: Matemática

Orientador: Doutor Amílcar dos Santos Costa Sernadas

Co-Orientador: Doutor Vasco Manuel Thudichum de Serpa Vasconcelos

Resumo:

Os objectos concorrentes podem oferecer os seus serviços de forma não uniforme devido a constrangimentos de sincronização, ou seja, a disponibilidade dos serviços depende do estado interno do objecto. Interfaces-como-tipos é uma abordagem que não se adequa à modelação da disponibilidade não uniforme de serviços, e o erro de comunicação "*message-not-understood*" é muito restricto. Por isso propõe-se uma definição mais abrangente de erro de comunicação: exige-se apenas que mensagens que solicitam um método desactivado, mas que podem ser aceites num dado momento futuro, não provoquem erros. Esta nova noção detecta mensagens que nunca podem ser aceites, quer porque o método solicitado não existe de todo, quer porque o objecto está bloqueado e não pode mudar de estado para aceitar a solicitação. Formalizam-se objectos concorrentes não uniformes em TyCO, um cálculo de objectos com passagem de nomes, e garante-se a segurança de programas (a ausência de erros em tempo de execução) recorrendo a um sistema de tipos. Os tipos são termos de uma álgebra de processos que descrevem alguns aspectos dinâmicos do comportamento dos objectos, correspondendo a uma abordagem de comportamentos-como-tipos. O sistema de tipos atribui termos da álgebra aos nomes dos processos, e garante que processos tipificáveis nem estão localmente bloqueados, nem geram erros em tempo de execução.

Palavras-chave: Orientação a Objectos; Concorrência; Álgebra de Processos; Bissimulação; Tipos; Sistema de Tipos.

Typing Non-uniform Concurrent Objects

Abstract

Concurrent objects can offer services non-uniformly, according to synchronisation constraints, that is, the availability of a service depends on the internal state of the object. Interfaces-as-types is an approach not suitable to model non-uniform service availability, and the message-not-understood communication error is very restrictive. Hence, we propose a looser definition of communication error by demanding only that messages requesting a method not enabled, but that may be accepted at some time in the future, do not cause errors. This new notion detects messages that are never accepted, either because the requested method does not exists at all, or because the object is blocked and cannot change its state to accept the request. We formalise non-uniform concurrent objects in TyCO, a name-passing object calculus, and we ensure program safety (the absence of run-time errors) via a type system. Types are terms of a process algebra which describe some dynamic aspects of the behaviour of objects, thus, the approach is centred on behaviours-as-types. The type system assigns terms of the algebra to the names of processes, and ensures that typable processes are neither locally blocked nor run into communication errors.

Key-words: Object-orientation; Concurrency; Process Calculus; Bisimulation; Types; Type System.

Acknowledgements

Special thanks are due to: Vasco Vasconcelos, co-supervisor of this thesis, for his continuous support, rigorous critics, and numerous suggestions; Amílcar Sernadas, my supervisor, responsible for my graduate education, and for my discover of the joys of research; Pedro Resende who, without any formal role, conducted me in the realm of process algebras.

I would also like to mention Luís Lopes, with whom I discussed many aspects of this thesis. In particular, Chapter 4 results from joint work.

The stimulating and friendly working environment of the Computer Science group of the Mathematics Department was invaluable. Thanks are due to all past and present members.

The *Wednesday Morning Club* of the Department of Informatics of the Lisbon Faculty of Sciences was a very gratifying experience. I had several opportunities of presenting there various stages of my work, profiting from important comments. Thanks are due to all participants.

Uwe Nestmann and Gérard Boudol hosted me in very important periods. I thank both for the invitation, the scientific support, and the friendship. Thanks are also due to the Ny-Klubben at the Computer Science Department of The Aalborg University, and to the Mobility group at INRIA Sophia-Antipolis, for the warm welcome, as well as for very important suggestions and fruitful discussions.

Jean-Louis Colaço was an exigent reader and made various important comments. I am grateful to him, also for for the invitation to visit Toulouse. Marc Pantel and Patrick Sallé deserve a special mention, for their support.

The Danish institute BRICS, the ENSEEIHT at Toulouse, and the Franco-Portugais project '*Sémantique des objets concurrents*' funded my visits. APPIA funded the participation in summerschools. This work was also partially supported by the Portuguese *Fundação para a Ciência e a Tecnologia*, the PRAXIS XXI Projects 2/2.1/TIT/1658/95 LogComp and PRAXIS/P/EEI/120598/98 DiCoMo, as well as by the ESPRIT IV Working Group 22704 ASPIRE.

Finally, I am deeply grateful to Dina, who always supported me, sacrificing herself many times.

Contents

Part I

Introduction

Chapter 1

Rationale

Over the last twenty years, one of the most popular programming paradigms has been that of *object-orientation* [Weg90]. Objects are abstract entities that describe computational components through their behaviour and state. They constitute the basic building blocks of programs and systems. Within this paradigm, not only have several languages emerged and stabilised — like SIMULA [DN67, ND81] and SMALLTALK [GR83], but being C++ [Str91] and JAVA [GJS96] the more widely used among them — but also a very large body of research has been formed. Nowadays, object-oriented notions, as abstraction, encapsulation, or inheritance, play a central role in software and hardware development. Basic research has made important contributions, providing a firm theoretical ground for object-oriented programming, and shedding light on several cumbersome aspects of the paradigm. In particular, the λ-calculus is widely accepted as a basis for studying features like formal semantics or type systems for sequential object-oriented languages.

A system composed by several components that share resources and interact to perform certain tasks, is a *concurrent system*. Communication is the key feature of a concurrent (possibly distributed) system. The interaction between components — or *processes* — of the system is modelled by either *synchronous* communication — a "hand-shake" mechanism — or by *asynchronous* communication. Message passing is a natural way of modelling process interaction in a concurrent system, in particular when such system is not equipped with a global clock. The main characteristic of asynchronous communication is that the sender of a message cannot know when the message is actually received. Since this form of communication does not require a rendez-vous between the sender and the receiver, it is easier to implement it in a distributed system, using only the system's primitives.

Concurrent scenarios increase the need for theoretical foundations. The behaviour of an object is no longer linear, but branched, which makes much harder the formal verification that a property holds for some program. A simple mathematical model is crucial to successfully represent and study these systems. A very simple model — yet rich and usually computationally complete — is that of *process algebra*. There are several proposals, but all consider communication to be the central aspect of a concurrent system, and represent systems by composing a few, very simple, operators, including:

1. atomic actions, often regarded as communication along channels;
2. parallel composition;
3. sequencing;
4. non-deterministic choice.

All these features are present in process algebras like ACP, the *Algebra of Communicating Processes* of Bergstra an Klop [BK85, BW90], CSP, the *Communicating Sequential Processes* of Hoare [Hoa85], and CCS, the *Calculus of Communicating Systems* of Milner [Mil89a]. Data structures are easily defined using the basic constructors, and values of the data structures can be send along the communication channels. In all, the topology of communication is fixed by the syntax. Furthermore, channels can be declared local to a process. When a system evolves, the topology may become simpler, if some channels are no longer used and hence, discarded, but new channels cannot be created.

Name-passing process calculi are process algebras where the only data allowed to circulate in the channels is channel names. However, they provide an operator to create new names within a given scope, and via communication, this scope can also be enlarged. Therefore, since the topology of communication can grow or shrink, name-passing calculi model mobile systems. The paradigm of such calculi is the π-calculus of Milner, Parrow, and Walker [MPW92], a *calculus of mobile processes*. The π-calculus is strictly more expressive than the ordinary process algebras, being able not only of representing all the systems and data structures that the former do, but also more, in particular mobile systems and higher-order processes.

Ideas, concepts, and techniques from (name-passing) process calculi have been successfully applied to the study of behavioural properties and type systems of concurrent object-oriented languages. On one hand, its extra features, like referencing and scoping, make process calculi approaches very suitable for studying and representing object-oriented programming. On the other hand, process calculi provide: (1) structural operational semantics — an essential element for reasoning about the correctness of programs; (2) various static type systems, ensuring the absence of run-time errors in well-typed programs; and (3) several notions of behavioural equivalences together with proof techniques, algebraic laws, and logical characterisations.

Process calculi theory is the main tool in this thesis. It is applied to model objects, abstracting their behaviour, and to develop a sound type system. If a motto were to be given, it would be *behavioural semantics as processes*.

1.1 Semantics of objects as processes

Modelling objects as processes is an important line of research in the process calculi community. It consists of translating (or encoding) an object language in a process calculus. Defining an operational, denotational, or axiomatic semantics for an high-level object-oriented language can be a difficult task. Moreover, once the job done, it may still be hard to perform proofs. A translation defines a semantics for the higher level language in terms of the lower level calculus. Results and proof techniques of the calculus can then be used to reason about the language. Obviously, this encoding must possess certain properties. A "good" encoding: (1) is compositional — encoding an expression constructed with a given operator gives the composition by the translation of the operator of the encodings of the operands; (2) is divergence-free; and (3) preserves and reflects operational and behavioural properties of both languages.

A pioneer work was that of Vaandrager [Vaa86], who encoded a language of the POOL-family [Ame89] into ACP. Nevertheless, object-oriented concepts like object identity and scoping are more naturally modelled in name-passing process calculi. The π-calculus has been used as the setting for similar studies. A brief account of these follows.

1. Walker [Wal95] was the first to use the π-calculus to give semantics to a parallel object-oriented language, proving a correspondence between this semantics and a transitional semantics of the language;
2. Liu and Walker [LW98] and Philipou and Walker [PW98] study the correctness of transformation rules that increase the concurrency of systems;
3. Zhang and Potter [ZP97] define a class-based model in the π-calculus, capturing method sharing and multiple inheritance.

Instead of using directly π-calculus, another possible approach is that of extending the calculus with objects, i.e. defining a process calculus with objects built in. The main examples of the use of this approach are outlined below.

1. Honda defines the ν-calculus [HT91], a calculus of objects that is an asynchronous version of the π-calculus. The ν-calculus turns out to be a subcalculus of π that preserves its computational power.
2. Nierstrasz defines OC [Nie92], a calculus of objects that unifies π and λ-calculi, supporting communication of tuples of terms;
3. Vasconcelos defines TyCO [VT93], Typed Concurrent Objects, a name-passing object calculus with asynchronous communication. The calculus is an extension of the asynchronous π-calculus with input-guarded labelled sums denoting methods in objects, and labelled asynchronous messages that select a branch of the sum. The aim of the work is to provide an operational semantics and a polymorphic type system that detects message-not understood errors, and the work includes a type inference algorithm.

4. Sangiorgi defines π^{V} [San98], a typed π-calculus that communicates *variant values*, constructed from names and variant tags, and includes a `case` constructor to decompose the variant values. The aim is the study of type systems, objects, and subtyping in a π-calculus .
5. Boudol defines the Blue Calculus [Bou98], a program passing calculus directly incorporating the λ-calculus into the π-calculus. The aim is a suitable programming notation for higher-order concurrency. He proposes a type system that encompasses both Curry's type inference for the λ, and Milner's sorting for the π. Dal-Zilio studies thoroughly type systems and objects in Blue [Dal99].

A very successful model of objects is Abadi and Cardelli's ς-calculus [AC96], actually a family of several typed calculi all based on a common untyped calculus. To import its various type systems into the π-calculus, Hüttel, Kleist and Sangiorgi define encodings [HK96, Kle00, San98]. DiBlasio and Fisher [DF96], and Gordon and Hankin [GH98] choose a different path, adding concurrent constructors to the ς-calculus.

Another setting where objects are studied as processes are *Actors* [HBS73, Agh86]. Actors have unique identities and communicate by asynchronous message passing; message reception triggers the execution of a method that may change the behaviour of the actor and that may lead to the creation of new actors. In a community of communicating actors a fairness assumption is assumed.

A different approach is that of Sernadas *et al.* that investigate the algebraic specification and verification of concurrent, object-oriented systems, using categorical techniques [CSSE92, CSS93, CSS94, ES90, EGS91, EC00, SSE87, SFSE89, SEC90, SE91, SSC95, SSC98].

1.2 Non-uniform objects

In the beginning of the 90's, Nierstrasz argued that typing concurrent objects posed particular problems, due to the '*non-uniform service availability of concurrent objects*'. By synchronisation constraints, the availability of a service depends upon the internal state of the object (which reflects the state of the system) [Nie95]. Objects exhibiting methods that may be enabled or disabled, according to their internal state, are very common in object-oriented programming, e.g. a stack — from which one cannot pop elements if it is empty, a finite buffer — where one cannot write if it is full, a cash machine — from which one can only get a balance if the connection with the bank is enabled.

Therefore, a static notion of typing, like interfaces-as-types, is not powerful enough to capture dynamic properties of the behaviour of concurrent objects. A rigid interface, exhibiting all the methods of an object, gives misleading information about the functionality of the object. Accordingly, Nierstrasz proposes the use of a regular language as types for active objects to characterise the traces of menus offered by the objects. He suggests a notion of subtyping, '*request substitutability*', based on a generalisation of the '*principle of substitutability*' by Wegner and Zdonick [WZ88]: services may be refined as long as the original promises are still upheld. Request substitutability follows the extension relation of Brinksma *et al.* [BSS87], and it is a transition relation close to the failures model.

Despite having developed a calculus of objects, Nierstrasz did not apply these ideas in a type system for it, neither did show how to model non-uniform objects. Some work has recently been done on these directions. A brief account follows. Thorough comparison with our work can be found in Chapter 15 on related work.

1. Colaço [CPS97, Col97] defines CAP, a *Calculus of Primitive Actors*[1] that extends TyCO with an operator to change the behaviour of objects, thus supporting objects with dynamically changing service offers. The aim is to develop a type system to detect some "orphan" messages, and the work includes a type inference algorithm.
2. Nimour and Najm [NN97] define COB, a *Calculus of Object Bindings* that also features objects with dynamically changing service offers, and that embodies a type system to detect "orphan" messages. The calculus is compliant with the Reference Model for Open Distributed Processing, ODP [NS97].
3. Puntigam [Pun97, Pun99] defines *Process Types*, a calculus of objects and types to specify (non-regular) sequences of messages. A static type system guarantees that a server understands all messages sent by concurrent clients, albeit the set of acceptable messages changes dynamically.

The π-calculus is a well established model of concurrency, with a very rich theory. We would like to study non-uniform service availability of concurrent objects in a calculus with objects built-in, but as close to π as possible, so that good mutual encodings would be feasible. We consider that TyCO is simpler than the proposals above, closer to π, and expressive enough to model non-uniform objects and to support type systems adequate for them. The next chapter gives an account of our work and contributions.

[1]French acronym: Calcul d'Acteurs Primitifs

Chapter 2

What is achieved

We aim at a more flexible style of programming and at a type discipline that deals with non-uniform concurrent objects. Our proposal is the following:

The interface of an object describes only those methods that are enabled; when a client asks for a disabled method the message is not rejected if the object may evolve to a state where the method becomes available.

To achieve these objectives, we propose a looser view of communication errors: an object-message pair is not a communication error if the message may be accepted at some time in the future. Therefore, an error-free process guarantees that messages are given a chance of being attended, since a permanently enabled object eventually receives a particular message target to it. In conclusion:

Errors are those processes containing an object that persistently refuses to accept a given message, either because the object is blocked, or because it will never have the right method for the message.

2.1 Aims

To ensure program safety, that is, the absence of run-time errors, a popular approach is the use of a static type system. Traditional type systems assign rigid interface-like types to the names of the objects [Mil93, VT93, KY95]. We adopt a behaviours-as-types approach, such that a type is able to characterise all possible life-cycles of an object, thus providing information about possible sequences of interactions. Types are terms of a process algebra, namely higher-order labelled transition systems, which describe some dynamic aspects of the behaviour of objects. These types are partial specifications of the behaviour of objects, able to cope with non-uniform service availability. They are also suitable for specifying communication protocols, like POP3, since a type can represent, in a branching structure, sequences of requests to a server.

Equipped with the more flexible notion of error mentioned before, and with these richer types, we develop a type system that assigns terms of the algebra to the names of processes, and ensures safety properties resulting from the absence of persistent bad redexes, guaranteeing that typable processes neither contain blocked objects, refusing to accept requests, nor will run into communication errors.

2.2 Outline of the thesis

The purpose of this thesis is the definition of a theory of types and of a type system to cope with non-uniform objects. It is composed by three main parts:

1. the formalisation of non-uniform objects in TyCO;
2. the development of a process algebra of behavioural types; and
3. the definition of a type system correct with respect to the notion of communication errors.

The paragraphs below give a detailed account of what is accomplished.

2.2.1 Typed concurrent non-uniform objects

Part II of this thesis is dedicated to non-uniform TyCO. The calculus naturally supports objects with dynamically changing service offers — non-uniform service availability — as the syntax does not force objects to own unique identities, or to display fixed interfaces. These are aspects that concern to a type system.

We impose a restriction on the calculus that is common in object calculi: "*do not receive on received names*". This condition, known as the *locality condition*, prohibits parameters of methods of being locations of objects. It simplifies the technical work, allows a richer theory, and has an object-oriented flavour, as objects can only create other objects within their scope.

Another aspect where we differ from other presentations of TyCO is that objects are non-deterministic and distributed. Input processes are regarded as views of (abstract) objects, thus one can compose in parallel several input processes at the same location and with different collections of methods. In what respects persistent objects, instead of replication or of process definition we introduce a recursive constructor.

Let us present the version of the calculus used herein. For simplicity, the constructors below are monadic, but TyCO is polyadic (i.e. communicates tuples of names). The basic process constructors are:

Labelled asynchronous messages $a!l[v]$, targeted to an object location a and carrying a labelled value $l[v]$.

Ephemeral objects $a?\{l_1(x_1) = P_1, \ldots, l_n(x_n) = P_n\}$, located at name a and offering methods $l_i(x_i) = P_i$ with formal parameter x_i and process P_i as body, where x_i does not occur as an object location in P_i.

Persistent objects $D[a_1, \ldots, a_l; v_1, \ldots, v_m]$ where

$$D \stackrel{\text{def}}{=} \mathbf{rec}\, X(x_1, \ldots, x_l; y_1, \ldots, y_m).x_1?\{l_1(z_1) = P_1, \ldots, l_n(z_n) = P_n\}\,.$$

The process variable X occurs in some method bodies and signals recursion points. The names in the sequences $a_1, \ldots, a_l$ are input-arguments, and in $v_1, \ldots, v_m$ are output-arguments; the instantiate the formal input-parameters $x_1, \ldots, x_l$ and the formal output-parameters $y_1, \ldots, y_m$. A class definition D is a template to generate potentially persistent objects, like

$$a_1?\{l_1(z_1) = P_1[D/X]\sigma, \ldots, l_n(z_n) = P_n[D/X]\sigma\}$$

where $\sigma \stackrel{\text{def}}{=} [a_1, \ldots, a_l; v_1, \ldots, v_m/x_1, \ldots, x_l; y_1, \ldots, y_m]$.

The remaining constructors are standard: *parallel composition*, $P\,|\,Q$, denoting processes P and Q running concurrently; *scope restriction*, $\nu x\, P$, denoting the restriction of the scope of a name x to process each P; and *inaction*, $\mathbf{0}$, denoting the terminated process.

Chapter 3 formally presents the syntax and operational semantics of this version of the calculus, and Chapter 4 presents non-trivial examples of non-uniform objects. The main contribution of this part is the dynamic notion of communication error-free processes that we call *persistent bad redexes*, which we define in Chapter 5.

2.2.2 The algebra of behavioural types

Part III of this thesis is dedicated to ABT, the Algebra of Behavioural Types. To capture the dynamic behaviour of a non-uniform object as a partial specification — a type — a process algebra is a natural choice. A type is basically a collection of enabled methods (an interface); such a type is dynamic in the sense that the execution of a method can change the type. We model disabled methods by means of a blocking operator, corresponding to an action external to the object. The basic type constructors are:

Labelled sums $\sum_{i\in I} l_i(\widetilde{\alpha}_i).\alpha_i$, the type of an object that offers a collection of methods $l_i(\widetilde{\alpha}_i)$ with parameters of types $\widetilde{\alpha}_i$, and that after the execution of method l_i changes its behaviour to α_i.

Blocked sums $\sum_{i\in I} \upsilon.\alpha_i$, the type of a blocked object. The invocation of a method in another object releases one of the possible behaviours α_i of the object.

Observe that the algebra is not based on communication. Actions are the methods of objects. The basic composition operator is the parallel, which is simply a merge of processes, denoting that there are several instances of an object in parallel. Persistent objects have infinite life-cycles, denoted by recursive types.

This part studies the semantic foundations of behavioural types for non-uniform concurrent objects, defining the syntax and the operational semantics of ABT, as well as an equivalence notion and its algebraic characterisation. The equivalence notion, label strong bisimulation or *lsb*, equates processes that offer the same methods, and after the execution of a method they remain equivalent, in a co-inductive style. This bisimulation notion is original and we claim that it is a natural one for types of (non-uniform) objects. We introduce an axiomatic system that is complete for image-finite types.

2.2.3 Type system

Part IV of this thesis is dedicated to the type system. Unlike in the λ-calculus where types are functional and assigned to terms, in name-passing process calculi types are assigned to names. The purpose of typing is avoiding arity mismatch when communicating tuples of names. Thus, types specify how the free names in a process carry other names in a safe way. Since in TyCO the input prefix is always guarding a labelled sum, it is also necessary to ensure that the output selects an existing method. Therefore, in Vasconcelos' work, communication errors coincide with the 'message-not-understood' error of object-oriented languages.

As explained before, we do not want to reject processes with occasional bad redexes. The purpose of the type system is to detect persistent bad redexes. The type of an object location, an input-type, specifies the behaviour of the object in a process in terms of the sequences of messages it can accept. The type of a message target, an output-type, specifies all the possible sequencings of the messages to an object that exist in a process. The type system is fairly simple, assigning typings, sets of triples — name, input-type, and output-type — to processes. To be sound with respect to the notion of error process, the typing assignment requires an extra checking: that input-types agree, or are compatible with, output-types. Agreement and compatibility are two transition relations on types that characterise error-free processes, that is, processes without persistent bad redexes: the former ensures that a process does not contain blocked objects; the latter, that a process does not contain messages that free objects never accept.

The verification of the two relations is done on minimal typings, i.e. the smallest typings that type check the process with the simplest types. The main results that the type system guarantee are:

1. Existence and uniqueness up to type equivalence of minimal typings.
2. Preservation of typability under computation — Subject Reduction — ensuring as a corollary that well-typed process do not run into errors.

Part II

A calculus of non-uniform concurrent objects

Chapter 3

Typed concurrent objects

TyCO — Typed Concurrent Objects — is a name-passing calculus featuring asynchronous communication between concurrent objects via labelled messages carrying names [VT93]. The calculus is an object-based extension of the asynchronous π-calculus [Bou92, HT91]; synchronous communication and the returning of results are implemented by means of "*reply-messages*". TyCO is reminiscent of the ς-calculus [AC96], the *Object Calculus* of Abadi and Cardelli, in the sense that objects are sums of labelled methods (the variable *self* being interpreted as the channel where the object is located), and messages can be seen as asynchronous method invocations. The ς-calculus has a method update operator that is not present in TyCO. Notice however that TyCO is a concurrent calculus and ς is not.

In TyCO the objects behave according to the principles of the actor model of concurrent computation [HBS73, Agh86], namely they communicate by asynchronous message passing, message reception triggers the execution of a method that may change the behaviour of the actor and that may lead to the creation of new actors. Nevertheless, the calculus differs from actors in that the principle of uniqueness of actors' names is not applied, and the fairness assumption is not assumed.

This chapter focuses on the aspects of TyCO which are particular to this thesis. For a more general and detailed presentation of the calculus see, among others, Vasconcelos's thesis [Vas94a], a recent paper [Vas99], or the TyCO implementation home page (URL: www.ncc.up.pt/∼lblopes/tyco).

Herein, we introduces the syntax and operational semantics of the version of TyCO that we use. We discuss several aspects of the calculus that are specific to this thesis, namely that several objects may share a location and that objects are non-deterministic, and compare the constructor that provides unbounded behaviour with other alternatives not considered here.

3.1 Syntax

Consider a countable set of *names* and a set of *labels*, disjoint, and further, assume a countable set of *process variables*, disjoint from the previous two sets.

3.1.1 A local calculus

We define herein *Local* TyCO, a restricted version of the calculus where the parameters of a method cannot be locations of objects in the body of a method.

NOTATION. Let us fix some notation.

1. Lower case letters a, b, v, x, y, possibly subscripted, denote *names*; $\tilde{v}$ stands for a sequence of names, and $\tilde{x}$ for a sequence of pairwise distinct names; moreover, $\vec{a}$ denotes a pair of sequences of names, $\tilde{a}\,;\tilde{b}$ with $\vec{a}_{\mathsf{i}}$ denoting $\tilde{a}$ and with $\vec{a}_{\mathsf{o}}$ denoting $\tilde{b}$.
2. Lower case letters $l, m, n, \ldots$, possibly subscripted, denote *labels*.
3. Capital letters, $X, Y, \ldots$, denote *process variables*.

The definition below establishes the syntax of TyCO processes.

DEFINITION 3.1.1 (PROCESSES).
The following grammar defines the set $\mathcal{P}$ of processes *where I is a finite indexing set.*

$$
\begin{array}{rlr}
P ::= & \mathbf{0} & \textit{terminated process} \\
\mid & a?M & \textit{ephemeral object} \\
\mid & (\mathsf{rec}\, X(\vec{x}).a?M)[\vec{v}] & \textit{persistent object} \\
\mid & a!l[\tilde{v}] & \textit{asynchronous message} \\
\mid & X[\vec{v}] & \textit{recursive call} \\
\mid & \nu x\, P & \textit{variable declaration} \\
\mid & P \mid Q & \textit{parallel composition} \\
M ::= & \{l_i(\tilde{x}_i) = P_i \mid i \in I\} & \textit{method collection}
\end{array}
$$

Objects, of the form $a?M$, and *messages*, of the form $a!l[\tilde{v}]$, are the basic processes in the calculus. An object is an input-guarded labelled sum where the name a is the *location* of the object. M is a finite collection of labelled methods; each method $l(\tilde{x}) = P$ in M is labelled by a label l, has a finite sequence of names $\tilde{x}$ as parameters, and has an arbitrary process P for body. In an asynchronous message the name a is the target; the message carries a *labelled value* $l[\tilde{v}]$ that selects the method l with arguments $\tilde{v}$. The process $(\mathsf{rec}\, X(\vec{x}).a?M)[\vec{v}]$ denotes a *persistent object*, i.e. an object with potentially unbounded behaviour. It is a recursive definition with a pair of sequences of parameters — $\vec{x}_{\mathsf{i}}$ denoting the locations of objects and $\vec{x}_{\mathsf{o}}$ denoting the targets of messages — together with two sequences of arguments to instantiate this parameters.[1] $X[\vec{v}]$ is a *recursive call.*

The remaining constructors of the calculus are fairly standard in name-passing process calculi: process $P \mid Q$ denotes the *parallel composition* of processes; process $\nu x\, P$ denotes the *scope restriction* of the name x to the process P (often seen as the creation of a new name, visible only within P); finally, *inaction*, denoted $\mathbf{0}$, is the terminated process.

TERMINOLOGY. Objects and messages are, respectively, the *input* and the *output* processes. An object location is often called the *subject of the input process*, or an *input name*; a message target is often called the *subject of the output process*, or an *output name*. The names that a message carries, often called the *objects of the output process*, are also output names. Moreover, $\vec{x}_{\mathsf{i}}$ are input names and $\vec{x}_{\mathsf{o}}$ are output names.

LOCALITY CONDITION. By technical reasons rose by the type system, we impose a syntactic restriction in the calculus. We henceforth assume that *only the output capability of names may be transmitted*, e.g. in a method $l(\tilde{x}) = P$ the parameters $\tilde{x}$ are not allowed to be locations of objects in the body P (i.e. we rule out processes like $a?\{l(x) = x?M\}$).

Such a restriction is present in object-oriented languages where the creation of a new name and of a new object are tightly coupled (e.g. JAVA), as well as in recent versions of the asynchronous π-calculus [Ama97, Bor98, MS98], and in the Join-calculus [FGL+97]. The authors claim that this restriction has an object-oriented flavour, since a name has a clear scope where all the inputs are defined, and outside that scope other objects can access its methods, but cannot create new objects with that name. This restriction is crucial; we would not know how to type non-uniform objects without it.

RECURSIVE DEFINITIONS. The complexity of the recursive constructor is a burden from a programming point of view, but it provides information that plays a fundamental role in the type system. The following example illustrates the behaviour of a recursive definition. For more thorough examples see the references on TyCO [VT93, Vas94a, Vas99] and the examples in the distribution of TyCO.

EXAMPLE 3.1.2 (ONE-PLACE CELL). cf. Example 2.5.5, [Vas94a].
A very simple example of a persistent object is the *one-place buffer cell*: it stores a value and offers two methods: *write*, with a parameter that is the new value to store; and *read*, with a parameter that is the return address.

$$(\textbf{rec}\ \mathsf{Cell}(c\,;v).c?\{\mathit{write}(u) = \mathsf{Cell}[c\,;u]\,,\ \mathit{read}(r) = r!\mathit{val}[v] \mid \mathsf{Cell}[c\,;v]\})[a\,;w]$$

[1] We have adapted this constructor from the asynchronous π_1-calculus [Ama97].

A *write* request to $\mathsf{Cell}[a\,;w]$ triggers a computation — defined ahead as reduction, $\longrightarrow$ — that stores the value b in the Cell, overwriting the previous value w: $\mathsf{Cell}[a\,;w] \mid a!write[b] \longrightarrow \mathsf{Cell}[a\,;b]$. A *read* request does not change the state of the cell, but simply sends to the client the values it stores: $\mathsf{Cell}[a\,;b] \mid a!read[r] \longrightarrow \mathsf{Cell}[a\,;b] \mid r!val[b]$.

NOTATION. To simplify the examples, we introduce some abbreviations in the syntax.

1. In a method like $l(\tilde{x}) = P$, when the sequence of parameters $\tilde{x}$ is empty we omit the symbol ε that denotes the empty sequence, as well as the parentheses, writing $l = P$ instead of $l(\varepsilon) = P$. Similarly, we omit the symbol ε when one of the sequences of parameters of a recursive definition is empty, like in $X[;\tilde{a}]$ or $X[\tilde{a};]$. Furthermore, we abbreviate to l a method $l = \mathbf{0}$ or a labelled value $l[]$.
2. We abbreviate to $\nu\tilde{x}\,P$ a process $\nu x_1 \cdots \nu x_n\,P$, and consider that the operator 'ν' extends as far to the right as possible and that it binds looser than the parallel composition operator '$|$'. This contrasts with the current practice in process calculi, but is along the lines of the programming languages, where the scope of a variable declaration is the block where it occurs.
3. Finally, $\{\tilde{x}\}$ denotes the set of names in $\tilde{x}$, $\{\vec{x}\}$ denotes $\{\vec{x}_\mathsf{i}\} \cup \{\vec{x}_\mathsf{o}\}$, and $x\vec{x}$ denotes $x\vec{x}_\mathsf{i};\vec{x}_\mathsf{o}$.

3.1.2 DISTRIBUTED AND NON-DETERMINISTIC OBJECTS

TyCO is a calculus, and thus should be as general as possible. Therefore, we only impose restrictions when they are necessary for technical reasons, as we are interested in studying the expressive power of the calculus. Most of the restrictions of an object-oriented setting, like the uniqueness of an object's name, can easily be imposed using a simple type system.

DISTRIBUTED OBJECTS. In contrast with the actor model, and with most object-oriented languages, in TyCO there can be several objects sharing the same location and locations without associated objects. Some authors defend that objects should have unique identities. That approach has drawbacks in an object calculus, namely it compromises composition. Therefore, an input process is a *view* of an abstract entity — "the object". We call *distributed object* a parallel composition of several input processes sharing the same location, possibly with different sets of methods, describing different copies — or clones — of the same object, each in a different state. The distribution is transparent to the user of an object. Take the process $a?\{m(\tilde{x}) = P\} \,|\, a?\{n(\tilde{y}) = Q\}$ where an object a offers methods m and n through two input processes with subject a. They are different clones of "the object a", each in a different state. An observer requesting one method cannot distinguish the behaviour of the above implementation of the object a from that of a "centralised" implementation like $a?\{m(\tilde{x}) = P \mid a?\{n(\tilde{y}) = Q\}, n(\tilde{y}) = Q \mid a?\{m(\tilde{x}) = P\}\}$. One recognises in this equivalence a particular form of Milner's expansion law [Mil89a].

NON-DETERMINISTIC OBJECTS. Further, notice that an object may contain several methods sharing a label, this leading to a non-deterministic behaviour. Thus, one can write objects like $a?\{l(\tilde{x}) = P, l(\tilde{y}) = Q\}$. Again, it is the expansion law that is responsible for this feature, as the process $a?\{l(\tilde{x}) = P\} \mid a?\{l(\tilde{y}) = Q\}$ is equivalent to the process $a?\{l(\tilde{x})= P \mid a?\{l(\tilde{y})= Q\}, l(\tilde{y})= Q \mid a?\{l(\tilde{x})= P\}\}$.

Objects with internal non-determinism are also easily defined, as it is very simple to program an internal choice operator, $\oplus$, as in $P \oplus Q \stackrel{\mathrm{def}}{=} \nu c\, c!m \mid c?\{m= P, m= Q\}$.

3.2 Operational semantics

Following the "chemical style" [BB92] and Milner *et al.* [MPW92], we define the operational semantics of TyCO via two binary relations on processes: a static one — *structural congruence* — and a dynamic one — a *labelled transition relation* — as this formulation expresses more naturally the behavioural aspects of the calculus that we seek. The relations use the notions of *free* and *bound* variables and names, and a notion of *substitution*.

DEFINITION 3.2.1 (BOUND VARIABLES AND NAMES, AND SUBSTITUTION).

1. *An occurrence of a process variable X in a process P is* bound*, if it occurs in the part M of a subterm $(\mathbf{rec}\, X(\vec{x}).a?M)[\vec{v}]$ of P; otherwise the occurrence of X is* free.
2. *An occurrence of a name x in a process P is* bound *if it occurs in the subterm Q of a part $l(\tilde{w}x\tilde{y})= Q$, or in the subterm Q of $\nu x\, Q$, or in the part M of the the subterm $(\mathbf{rec}\, X(\vec{x}).a?M)[\vec{v}]$ where $x \in \{\vec{x}\}$; otherwise the occurrence of x is* free.

   Accordingly, we define the set $\mathrm{fn}(P)$ *of the* free names *in P, the set* $\mathrm{bn}(P)$ *of the* bound names *in P, and the set* $\mathrm{n}(P) \stackrel{\mathrm{def}}{=} \mathrm{fn}(P) \cup \mathrm{bn}(P)$ *of the* names *in P.*
3. Alpha-conversion*, $\equiv_\alpha$, affects both bound names and bound process variables, and is defined as usual.*
4. *The process $P[\tilde{v}/\tilde{x}]$ denotes the* simultaneous substitution *in P of the names in $\tilde{v}$ for the free occurrences of the respective names in $\tilde{x}$; it is defined only when $\tilde{x}$ and $\tilde{v}$ are of the same length. Similarly, the process $P[\vec{v}/\vec{x}]$ denotes the* simultaneous substitution *in P of the pair of sequences in $\vec{v}$ for the respective sequences in $\vec{x}$. Finally, the process $P[A/X]$ denotes the* simultaneous substitution *in P of the part A for the free occurrences of the process variable X.*

Henceforth, we assume the standard name convention on names and process variables like in Barendregt [Bar84]. Furthermore, we consider that process variables occur only bound in processes, and that in a recursive definition, like $(\mathbf{rec}\, X(\vec{x}).a?M)[\vec{v}]$, the condition $\mathrm{fn}(a?M) \subseteq \{\vec{x}\}$ holds. Notice that $\{\vec{x}_\mathsf{i}\} \cap \{\vec{x}_\mathsf{o}\}$ is not necessarily empty; the sequences $\vec{x}_\mathsf{i}$ and $\vec{x}_\mathsf{o}$ can share names, as a parameter can occur in the object $a?M$ both as an input name and as an output name. A very simple static type system ensures these conditions, as well as the syntactic correctness of processes.

3.2.1 Structural congruence

The following relation defines a structural equivalence on processes. It allows (static) rearrangements of processes, fully preserving their behavioural properties.

Definition 3.2.2 (Structural congruence).
The following rules inductively define the structural congruence relation*.*

$$P \equiv Q\,,\ \textit{if}\ P \equiv_\alpha Q$$
$$a?M \equiv a?M' \ \textit{and}\ (\textbf{rec}\,X(\vec{x}).a?M)[\vec{v}] \equiv (\textbf{rec}\,X(\vec{x}).a?M')[\vec{v}]\,,\ \textit{if}\ M\ \textit{is a permutation of}\ M'$$
$$P \mid \mathbf{0} \ \equiv\ P \qquad P \mid Q \ \equiv\ Q \mid P \qquad (P \mid Q) \mid R \equiv\ P \mid (Q \mid R)$$
$$\nu x\,\mathbf{0} \equiv \mathbf{0} \qquad \nu xy\,P \equiv \nu yx\,P \qquad \nu x\,P \mid Q \equiv\ (\nu x\,P) \mid Q,\ \textit{if}\ x \notin \mathrm{fn}(Q)\,.$$

Processes are structurally equivalent up to α-conversion; the order of the methods in an object is not relevant, and neither is the order of restrictions; the set of processes equipped with parallel composition is an abelian monoid where inaction is the neutral element; restricting the scope of a name in an inactive process is irrelevant, as it is the order in which the names are restricted; the scope of a restriction can be enlarged to include a process Q involved in a parallel composition with that restriction, if the name restricted does not occur free in Q; similarly, the scope of a restriction that affects a parallel composition involving a process where name restricted does not occur free, can be diminished to release that process.

3.2.2 Labelled transition system

A *labelled transition relation* defines the computation mechanism of TyCO. In previous works the operational semantics are via a reduction relation, which is included in the labelled transition relation presented herein. Since the characterisation of the interacting behaviour of a process is a central aspect of this thesis, we opted for a labelled transition relation in an early style,[2] as it more naturally represents the interaction between a process and its environment.

Definition 3.2.3 (Message application).
The result of applying a labelled value $l[\tilde{v}]$ to a collection of methods M, denoted by $M \bullet l[\tilde{v}]$, is the process $P[\tilde{v}/\tilde{x}]$, if $l(\tilde{x}) = P$ is a method in M, and if the substitution is defined.

The basic action in TyCO is the execution of a method. The action is triggered by the reception of a message requesting the method, and results in the contraction of the redex composed by the object and the message — *message application*.

[2] Message reception — rules In and Rin — incorporates the instantiation of the parameters.

DEFINITION 3.2.4 (ACTION LABELS).
The following grammar defines the set of action labels *where* $\{\tilde{x}\} \subseteq \{\tilde{v}\} \setminus \{a\}$.

$$m ::= \tau \ \mid \ a?l[\tilde{v}] \ \mid \ \nu\tilde{x}\, a!l[\tilde{v}]\,.$$

The *silent action* τ denotes internal communication in the process; the *input action* $a?l[\tilde{v}]$ represents the reception on the name a of an l-labelled message carrying names $\tilde{v}$ as arguments; the *output action* $\nu\tilde{x}\, a!l[\tilde{v}]$ represents the emission to a of an l-labelled message carrying names $\tilde{v}$ as arguments, some of them bound (those in $\tilde{x}$; the name a is free to allow the message to be received). In the last two action labels, the name a is the *subject* of the label, and the names $\tilde{v}$ are their *objects*.

DEFINITION 3.2.5 (FREE AND BOUND NAMES IN LABELS). *An occurrence of a name x in an action label m is* bound, *if m is of the form $\nu\tilde{x}\, a!l[\tilde{v}]$ and $x \in \{\tilde{v}\} \cap \{\tilde{x}\}$; otherwise the occurrence of x is* free. *Accordingly, we define the sets* $\mathrm{fn}(m)$ *and* $\mathrm{bn}(m)$ *of the* free names *and of the* bound names *in an action label m, respectively.*

We define the operational semantics of TyCO via an *asynchronous transition relation* that is inspired by the labelled transition relation defined by Honda and Yoshida for the ν-calculus [HY95], and by that of Amadio, Castellani, and Sangiorgi for the asynchronous π-calculus [ACS98]. A detailed discussion on the relationship between both systems can be found in the work of Amadio *et al.* [ACS98].

DEFINITION 3.2.6 (LABELLED TRANSITION RELATION).
The rules of Table 3.1 inductively define the asynchronous transition relation.

The relation expresses the behaviour of a process in terms of its possible exchanges of messages with the environment, this characteristic emphasising the asynchronous nature of the relation, and thus, of the calculus. Notice that rules IN, RIN, COM, and REC assume that message application is defined. Otherwise, the transition does not take place. Furthermore, observe that, in rules PAR and RES, we assume the name convention.

REMARK. The *reduction relation* $\longrightarrow$ (cf. [VT93]) coincides with $\xrightarrow{\tau}$.

3.3 BASIC PROPERTIES OF THE TRANSITION RELATION

Some simple lemmas, relating transitions with the form of processes, will be useful to prove properties of the type system. The first shows that, during computation, free names are kept free and bound names are kept bound, or they may disappear. The second establishes a correspondence between the form of a process and its transitions.

(Out)	$a!l[\tilde{v}] \xrightarrow{a!l[\tilde{v}]} \mathbf{0}$	
(In)	$a?M \xrightarrow{a?l[\tilde{v}]} M \bullet l[\tilde{v}]$	
(Rin)	$(\mathsf{rec}\, X(x\vec{x}).x?M)[a\vec{u}] \xrightarrow{a?l[\tilde{v}]} (M[\mathsf{rec}\, X(x\vec{x}).x?M/X][a\vec{u}/x\vec{x}]) \bullet l[\tilde{v}]$	
(Com)	$a?M \mid a!l[\tilde{v}] \xrightarrow{\tau} M \bullet l[\tilde{v}]$	
(Rec)	$(\mathsf{rec}\, X(x\vec{x}).x?M)[a\vec{u}] \mid a!l[\tilde{v}] \xrightarrow{\tau} (M[\mathsf{rec}\, X(x\vec{x}).x?M/X][a\vec{u}/x\vec{x}]) \bullet l[\tilde{v}]$	
(Par)	$\dfrac{P \xrightarrow{m} Q}{P \mid R \xrightarrow{m} Q \mid R}$	
(Res)	$\dfrac{P \xrightarrow{m} Q}{\nu x\, P \xrightarrow{m} \nu x\, Q}$	$(x \notin \mathrm{fn}(m))$
(Open)	$\dfrac{P \xrightarrow{\nu\tilde{x}\, a!l[\tilde{v}]} Q}{\nu x\, P \xrightarrow{\nu x\tilde{x}\, a!l[\tilde{v}]} Q}$	$(a \notin \{x\tilde{x}\})$
(Struct)	$\dfrac{P \equiv P' \quad P' \xrightarrow{m} Q' \quad Q' \equiv Q}{P \xrightarrow{m} Q}$	

Table 3.1: The asynchronous transition relation of TyCO.

LEMMA 3.3.1 (PRESERVATION OF FREE AND BOUND NAMES).
Let $P \xrightarrow{m} Q$. *Then:*

1. $\mathrm{bn}(Q) \subseteq \mathrm{bn}(P)$, *and*
2. (a) *if* $m = a?l[\tilde{v}]$ *then* $\mathrm{fn}(Q) \subseteq \mathrm{fn}(P) \cup \{\tilde{v}\}$;

   (b) *if* $m = \nu\tilde{x}\, a!l[\tilde{v}]$ *then* $\mathrm{fn}(Q) \subseteq \mathrm{fn}(P) \cup \mathrm{bn}(m)$;

   (c) *if* $m = \tau$ *then* $\mathrm{fn}(Q) \subseteq \mathrm{fn}(P)$.

PROOF. By transition induction [Mil89a]. □

LEMMA 3.3.2 (STRUCTURAL CORRESPONDENCE).
Let $P \xrightarrow{m}$ *denote* $\exists_Q\, P \xrightarrow{m} Q$. *Then:*

1. $P \xrightarrow{\nu\tilde{x}\, a!l[\tilde{v}]}$, *if and only if,* $P \equiv \nu\tilde{x}\, a!l[\tilde{v}] \mid P'$;
2. $P \xrightarrow{a?l_j[\tilde{v}]}$, *if and only if,*

   either $P \equiv \nu\tilde{u}\, a?\{l_i(\tilde{x}_i) = P_i \mid i \in I\} \mid P'$

   or $P \equiv \nu\tilde{u}\, (\mathbf{rec}\, X(x\vec{x}).x?\{l_i(\tilde{x}_i) = P_i \mid i \in I\})[a\vec{v}] \mid P'$, *for some* $j \in I$ *such that* $\tilde{v}$ *and* $\tilde{x}_j$ *are of the same length, and* $a \notin \{\tilde{u}\}$;
3. $P \xrightarrow{\tau}$, *if and only if,*

   either $P \equiv \nu\tilde{u}\, a?\{l_i(\tilde{x}_i) = P_i \mid i \in I\} \mid a!l_j[\tilde{v}] \mid P'$

   or $P \equiv \nu\tilde{u}\, (\mathbf{rec}\, X(x\vec{x}).x?\{l_i(\tilde{x}_i) = P_i \mid i \in I\})[a\vec{v}] \mid a!l_j[\tilde{v}] \mid P'$, *for some* $j \in I$ *such that* $\tilde{v}$ *and* $\tilde{x}_j$ *are of the same length.*

PROOF. The 'only-if' direction is by transition induction. The 'if' direction is by constructing the derivation of the transition. □

3.3.1 PROCESS DECLARATION, RECURSION, AND REPLICATION

The original formulation of TyCO uses replication to provide for persistent objects. Despite its simplicity, replication is unwieldy. A recent formulation [Vas99] uses a form of process-declaration of which the recursive definition herein is a particular case. Process-declarations make mutual recursive definitions possible. The form of recursive definition we use is a simple one, which reduces to a single, input-guarded, process-declaration. Replication is a simple form of recursion where the recursive call occurs immediately after the first computation step. However, defining process-declaration only with recursion is cumbersome. A similar situation appears when comparing recursion with replication.

Encodings. Encodings of process definition into recursion, and of recursion into replication, are standard (see Vasconcelos thesis [Vas94a] and Parrow's introduction to the π-calculus [Par00]). They are homomorphic with respect to the simple processes — inaction, message and object, and compositional with respect to the composition operators — parallel and restriction.

We do not give the encodings herein, but illustrate them below through a well-known example: the one-place buffer.

Example. A one-place buffer is a simple object which only allows read operations when it is full, and write operations when it is empty. We like to specify it as follows, since it clearly shows the behaviour of the buffer, alternating between *write* and *read* operations.

$$\begin{aligned} P \stackrel{\text{def}}{=}\ & \textbf{def}\ \mathsf{Empty}(b) = b?\{\mathit{write}(u) = \mathsf{Full}[b\,;u]\} \\ & \textbf{and}\ \mathsf{Full}(b\,;u) = b?\{\mathit{read}(r) = r!\mathit{val}[u] \mid \mathsf{Empty}[b]\} \\ & \textbf{in}\ \mathsf{Empty}[b] \end{aligned}$$

Hence, $\mathsf{Empty}[b] \mid b!\mathit{write}[v] \mid b!\mathit{read}[r] \longrightarrow \mathsf{Full}[b\,;v] \mid b!\mathit{read}[r] \longrightarrow \mathsf{Empty}[b] \mid r!\mathit{val}[v]$.

Let Q be the translation of P into a process written with the recursive constructor. It is less readable, but exhibits the same observable behaviour than P.

$$Q \stackrel{\text{def}}{=} \nu\, e\, f\, R \mid e!\mathit{trig}[b] \text{ where}$$

$$R \stackrel{\text{def}}{=} (\textbf{rec}\, X(x\,;y).x?\{\mathit{trig}(b) = E \mid X[x\,;y]\})[e\,;f] \mid (\textbf{rec}\, Y(y\,;x).y?\{\mathit{trig}(b,v) = F \mid Y[y\,;x]\})[f\,;e]$$

$$E \stackrel{\text{def}}{=} (\textbf{rec}\,\mathsf{Empty}(z\,;w).z?\{\mathit{write}(u) = w!\mathit{trig}[z,u] \mid \mathsf{Empty}[z\,;w]\})[b\,;y]$$

$$F \stackrel{\text{def}}{=} (\textbf{rec}\,\mathsf{Full}(z\,;u).z?\{\mathit{read}(r) = r!\mathit{val}[u] \mid x!\mathit{trig}[z] \mid \mathsf{Full}[z\,;u]\})[b\,;v]$$

Like P, the process Q in the presence of requests for write and read behaves as follows:

$$Q \mid b!\mathit{write}[v] \mid b!\mathit{read}[r] \longrightarrow\longrightarrow \nu\, e\, f\, R \mid f!\mathit{trig}[b,v] \mid b!\mathit{read}[r] \longrightarrow\longrightarrow Q \mid r!\mathit{val}[v]\,.$$

A simpler recursive process still with the same observable behaviour is the following:

$$Q' \stackrel{\text{def}}{=} (\textbf{rec}\,\mathsf{Empty}(b\,;).b?\{\mathit{write}(u) = b?\{\mathit{read}(r) = r!\mathit{val}[u] \mid \mathsf{Empty}[b\,;]\}\})[b\,;]$$

Its translation into a replicated process P' is straightforward: $P' \stackrel{\text{def}}{=} \nu z\, z!\mathit{rec}[b] \mid R'$ where $R' \stackrel{\text{def}}{=} z?*\{\mathit{rec}(x) = x?\{\mathit{write}(u) = x?\{\mathit{read}(r) = r!\mathit{val}[u] \mid z!\mathit{rec}[b]\}\}\}$. Again, P' in the presence of requests for write and read behaves as expected:

$$P' \mid b!\mathit{write}[v] \mid b!\mathit{read}[r] \longrightarrow\longrightarrow \nu z\, Q \mid b?\{\mathit{read}(r) = r!\mathit{val}[v] \mid z!\mathit{rec}[b]\} \mid b!\mathit{read}[r] \longrightarrow\longrightarrow P' \mid r!\mathit{val}[v]\,.$$

It is important to notice that in all the cases the reductions are deterministic, that the write and read requests are taken in the same order, and that the resulting processes are the empty buffer and the result of a reading request taken after writing the value v.

Chapter 4

What are non-uniform objects, and what are they good for?

Concurrent objects do not always offer the same collections of methods, constraining the acceptance of messages on the states of objects. Due to this non-uniform service availability, noticed by Nierstrasz [Nie95], we advocate non-uniform interfaces, i.e. interfaces containing only enabled methods. Safe programs must guarantee that every message has a chance of being received if it requests a method that may become enabled at some point in the future. When a client asks for a disabled method, the message is not rejected if the object may evolve to a state where the method becomes available. This setting requires a looser view of communication errors: an object-message pair is not a communication error if the message may be accepted at some time in the future. Errors are those processes containing an object that persistently refuses to accept a given message, either because the object is blocked, or because it will never have the right method for the message.

Therefore, from the programming point of view, the use of non-uniform interfaces for representing objects, as opposed to the uniform record interfaces, allows a more intuitive and expressive definition of objects and, in general, of object-based data structures. This is a result of the far richer behaviours allowed by the non-uniformity. Furthermore, from an implementation point of view, the code is frequently much shorter, both in the source programs as well as in the final byte code or other executable format; even in cases where the difference in code size is not significant, the use of a single name for all of the object dynamic instances, which do not always have the same interface, substantially reduces heap space, and allows optimisations in heap usage.

The subsequent section presents examples of uniform and non-uniform code for several applications; for readability sake, the examples use the process declaration construct, which can be transformed into recursive definitions according to the encoding mentioned in Section 3.3.1.

4.1 A SIMPLE EXAMPLE: THE ONE-PLACE BUFFER

Take the example of the one-place buffer of Section 3.3.1, which only allows read operations when it is full, and write operations when it is empty. Recall its non-uniform code.

$$\begin{array}{l} \textbf{def } \mathsf{Empty}(b) = b?\{\mathit{write}(u) = \mathsf{Full}[b,u]\} \\ \textbf{and } \mathsf{Full}(b,u) = b?\{\mathit{read}(r) = r!\mathit{val}[u] \mid \mathsf{Empty}[b]\} \\ \textbf{in } \mathsf{Empty}[b] \end{array}$$

Traditional type systems, i.e. those that assign fixed records to names, reject the example above, since name b does not have a single interface containing both labels *write* and *read*. An alternative typable implementation, uses the busy-waiting technique to handle non-available operations.

$$\begin{array}{ll} \textbf{def} & \mathsf{Buf}(b,v,\mathit{empty}) = \\ & \quad b?\{\ \mathit{write}(u) = \textbf{if } \mathit{empty} \textbf{ then } \mathsf{Buf}[b,u,\mathit{false}] \\ & \qquad\qquad\qquad\quad \textbf{else } b!\mathit{write}[u] \mid \mathsf{Buf}[b,v,\mathit{false}] \\ & \qquad\quad \mathit{read}(r) = \textbf{if } \mathit{empty} \textbf{ then } b!\mathit{read}[r] \mid \mathsf{Buf}[b,v,\mathit{true}] \\ & \qquad\qquad\qquad\quad \textbf{else } r!\mathit{val}[v] \mid \mathsf{Buf}[b,v,\mathit{true}]\} \\ \textbf{in} & \mathsf{Buf}[b,v,\mathit{true}] \end{array}$$

In the second implementation, a process containing the redex $\mathsf{Buf}[b,v,\mathit{true}] \mid b!\mathit{read}[r]$ is not an error, as the *read* message can always be accepted, although not really processed, as the redex is restored. Only after a reduction involving a message $b!\mathit{write}[u]$ and $\mathsf{Buf}[b,v,\mathit{true}]$ may the *read* message be processed.

Similarly, in the first implementation, a process containing the redex $\mathsf{Empty}[b] \mid b!\mathit{read}[r]$ should not be considered an error, as again, the presence of a message like $b!\mathit{write}[u]$ also makes the reception of the *read* message possible. Nonetheless, notice that a deadlocked process like $\nu b\, \mathsf{Empty}[b] \mid b!\mathit{read}[r]$ should be considered an error.

In conclusion, from the point of view of an external observer, the implementations behave similarly, in the sense that both only process a *read* message when the buffer is full, afterwards becoming empty, and only process a *write* message when the buffer is empty, afterwards becoming full. However, the first implementation more naturally expresses the buffer's behaviour.

This style of programming is state-dependent, that is, for each object we specify its different interfaces according to the availability of services. The example presented can easily be scaled-up to an n-place finite buffer, by introducing a third state where the buffer is neither empty nor full. Objects usually have a finite (and small) number of such states, thus it is not difficult to program in this style. Moreover, an automatic translation of an uniform object into a non-uniform one is not an easy task, since it requires a specification of the object's behaviour.

4.2 Scaling-up: more complex examples

Consider an Automatic Teller Machine, ATM, an FTP server, and a POP3 Server. The first is a typical non-uniform object, as the services provided depend on various synchronisation constraints: the status of the machine (online or off-line), the type of the user's card, the kind of service the user requests, etc. The other two illustrate how non-uniform implementations require less synchronisation code, and allow optimisations in heap usage. We present them all below, each in non-uniform and uniform implementations.

4.2.1 ATM

This example is inspired on a similar one by Honda *et al.* [HVK98]. The ATM is a definition with two parameters, its address, a, and the bank's address, b. A request for a session with the ATM creates a new thread which establishes a private communication between the user and the ATM in name s. This private object recurrently offers a collection of methods: deposit, withdraw, and balance, together with an option for finishing the session. Each request results in a change in the state of the object, which presents a different interface for each of its subsequent states.

Non-uniform implementation. One private address for communication between the user and the ATM suffices. It is the address of the object that offer the ATM's services. Notice that the interfaces are state dependent, and that the reply messages do not always have the same aspect.

```
def Atm(a,b) =
      a?{req(r) = new s r!rep[s] | Atm[a,b] | Action[s,b,r]}

and Action(s,b,r) =
      s?{deposit  = Dep[s,b,r],
         withdraw = Get[s,b,r],
         balance  = Bal[s,b,r],
         quit     = r!bye}

and Dep(s,b,r) =
      s?{amount(u) = new v b!dep[u,v] | Rep[v,s,b,r],
         back      = Action[s,b,r]}

and Rep(v,s,b,r) =
      v?{done = r!ok | Action[s,b,r],
         fail = r!sorry | Action[s,b,r]}
```

```
and Get(s,b,r) =
      s?{five      = Wit[5,s,b,r],
         ten       = Wit[10,s,b,r],
         twenty    = Wit[20,s,b,r],
         amount(u) = Wit[u,s,b,r],
         back      = Action[s,b,r]}

and Wit(u,s,b,r) = new v b!get[u,v]  | Rep[v,s,b,r]

and Bal(s,b,r) = new v b!val[v] |
                 v?{done(u) = r!ok[u] | Action[s,b,r],
                    fail    = r!sorry | Action[s,b,r]}
```

From the user's point of view, a possible session would be as follows: deposit some amount (say, 17), then withdraw (say, 10), and finally get a balance. Observe that the synchronisation happens always on the user's name.

```
def Usr(q,a) = new u a!req[u] |
      u?{rep(s) = s!deposit | s!amount[17] |
                  u?{ok    = s!withdraw | s!ten |
                             u?{ok    = s!balance |
                                        u?{ok(q) = Rel[s,u,q,a],
                                           sorry = Rel[s,u,q,a]},
                                sorry = Rel[s,u,q,a]},
                     sorry = Rel[s,u,q,a]}}

and Rel(s,u,q,a) = s!quit | u?{bye = Usr[u,q,a]}
```

Uniform implementation. After each request of a method, the user has a new private address to communicate with the ATM, since the same name cannot have several interfaces. An alternative would be to have a single interface, with all methods, whether they were enabled or not.

```
def Atm(a,b) =
      a?{req(r) = new s r!rep[s] | Atm[a,b] | Action[s,b]}

and Action(s,b) =
      s?{deposit(r)  = new z r!rep[z] | Dep[z,s,b],
         withdraw(r) = new z r!rep[z] | Get[z,s,b],
         balance(r)  = Bal[s,b,r],
         quit(r)     = r!bye}
```

```
and Dep(z,s,b) =
      z?{amount(u,r) = new v b!dep[u,v] | Rep[v,s,b,r],
         back(r)     = r!ok | Action[s,b]}

 and Rep(v,s,b,r) =
       v?{done = r!ok | Action[s,b],
          fail = r!sorry | Action[s,b]}

 and Get(z,s,b) =
       z?{five(r)     = Wit[5,s,b,r],
          ten(r)      = Wit[10,s,b,r],
          twenty(r)   = Wit[20,s,b,r],
          amount(u,r) = Wit[u,s,b,r],
          back(r)     = r!ok | Action[s,b]}

 and Wit(u,s,b,r) = new v b!get[u,v]  | Rep[v,s,b,r]

 and Bal(s,b,r) = new v b!val[v] |
                  v?{done(u) = r!ok[u] | Action[s,b],
                     fail    = r!sorry | Action[s,b]}
```

From the user's point of view, a possible session would now be as follows. Observe that each request demands a new name for synchronisation, in a total of five extra names.

```
def Usr(q,a) = new u a!req[u] |
     u?{rep(s) = new r1 s!deposit[r1] |
          r1?{rep(z) = new r2 z!amount[10,r2] |
               r2?{ok = new r3 s!withdraw[r3] |
                    r3?{rep(z) = new r4 z!ten[r4] |
                          r4?{ok = new r5 s!balance[r5] |
                               r5?{ok(q) = Rel[s,u,q,a],
                                   sorry = Rel[s,u,q,a]},
                              sorry = Rel[s,u,q,a]}},
                        sorry = Rel[s,u,q,a]}}}

and Rel(s,u,q,a) = s!quit | u?{bye = Usr[u,q,a]}
```

4.2.2 FTP server

A rendez-vous with an FTP server triggers a new thread *b* for private communication with the client. The session starts with the client sending a username and a valid password. Then the client receives a new private channel to communicate with the server, and through that channel, can get files from, and put files in, the server. Each command requires a reply-to channel.

Non-uniform implementation. There is single thread for private communication with the client, although the interfaces of each state are different. Notice that several synchronisation steps are saved, and the channel to communicate with the FTP daemon is always the same. Consider available a name server, located at *nis*.

```
def Init(pid) = new b Ftpd[pid,b] | FtpThread[b]

and Ftpd(pid,b) =
     pid?{rv(r) = new s r!ok[s] |
                  s?{val(uid,pas) = b!thr[uid,pas,s,r] | Ftpd[pid,b]}}

and FtpThread(b) =
     b?{thr(uid,pas,s,r) = new x nis!rv[x] |
                           x?{ok(j) = j!ckusr[uid,pas,x] |
                                      x?{invalid = r!sorry,
                                         valid   = r!welcome |
                                                   Action[s,b,r]}}}

and Action(s,b,r) =
     s?{get(f) = r!send[f] | Action[s,b,r],
        put(f) = r!done | Action[s,b,r],
        bye    = FtpThread[b]}
```

From the user's point of view, a possible session would be to put a file and get another one.

```
def Clt(uid,pid) =
     new r,pas,f pid!rv[r] |
     r?{ok(s) = s!val[uid,pas] |
                r?{welcome = s!put[f] |
                             r?{done = s!get[f] |
                                       r?{send(f) = s!bye |
                                                    Clt[uid,pid]}},
                   sorry   = Clt[uid,pid]}}
```

Uniform implementation. Each step of the protocol requires a new reply-to channel, and creates a new private name to communicate with the client, as the interfaces are always different.

```
def Init(pid) = new b Ftpd[pid,b] | FtpThread[b]

and Ftpd(pid,b) =
     pid?{rv(r) = new s r!ok[s] |
          s?{val(uid,pas,r) = b!thr[uid,pas,r] | Ftpd[pid,b]}}

and FtpThread(b) =
     b?{thr(uid,pas,r) = new x nis!rv[x] |
                         x?{ok(j) = new y j!ckusr[uid,pas,y] |
                                    y?{invalid = r!sorry,
                                       valid   = new a r!welcome[a] |
                                                 Action[a,b]}}}

and Action(a,b) = a?{get(f,r) = r!send[f] | Action[a,b],
                     put(f,r) = r!done | Action[a,b],
                     bye      = FtpThread[b]}
```

From the user's point of view, a possible session would now be as follows, requiring four more names.

```
def Clt(uid,pid) = new r1,pas,f pid!rv[r] |
     r1?{ok(s) = new r2 s!val[uid,pas,r2] |
                r2?{welcome(a) = new r3 a!put[f,r3] |
                                 r3?{done = new r4 a!get[f,r4] |
                                            r4?{send(f) = a!bye |
                                                          Clt[uid,pid]}},
                    sorry      = Clt[uid,pid]}}
```

4.2.3 POP3 server

A POP3 server also requires an authorisation protocol, similar to that of the FTP server. The session starts after the client has sent first its username, then a valid password. If this process succeeds, the client can retrieve its mail from the server.

The implementation below is quite simplified. We present it here because the readability of the protocol really benefits from a non-uniform perspective (particularly the types), but from a code size point of view the non-uniform code is larger that the uniform, since the definitions require more parameters.

Non-uniform implementation. Similar to the respective version of the FTP server. Notice that the communication protocol is always performed along the same channel b, using only one name, r, for synchronisation.

```
def POP3d(pid) =
      pid?{rv(r) = new b r!ok[b] | Auth[b,r] | POP3d[pid]}

and Auth(b,r) =
       b?{user(uid) = Auth1[b,r,uid],
          quit      = Update[b,r]}

and Auth1(b,r,uid) =
       b?{user(newuid) = Auth1[b,r,newuid],
          pass(pas)    = new w nis![w] |
                         w?{ok(s) = s!chkusr[uid,pas] |
                                    s?{invalid = r!sorry,
                                       valid   = r!ok | Trans[b,r]}},
          quit         = Update[b,r]}

and Trans(b,r) =
      b?{stat    = r!done | Trans[b,r],
         list(k) = r!done | Trans[b,r],
         retr(k) = r!done | Trans[b,r],
         rset    = r!done | Trans[b,r],
         del(k)  = r!done | Trans[b,r],
         noop    = r!done | Trans[b,r],
         quit    = Update[b,r]}

and Update(b,r) = -- update state after transactions | r!bye
```

Uniform implementation. Similar to the respective version of the FTP server. In each different state of the protocol the communication is done through a different channel and always with new names for synchronisation.

```
def POP3d(pid) =
      pid?{rv(r) = new b r!ok[b] | Auth[b] | POP3d[pid]}

and Auth(b) =
      b?{user(uid,r) = new s r!ok[s] | Auth1[s,uid],
         quit(r)     = Update[b,r]}
```

```
and Auth1(s,uid) =
      s?{user(newuid,r) = new s r!ok[s] | Auth1[s,newuid],
         pass(pas,r)    = new w nis![w] |
                          w?{ok(s) = new s1 s!chkusr[uid,pas,s1] |
                                     s1?{invalid = r!sorry,
                                         valid   = new z r!ok[z] |
                                                   Trans[z]}},
         quit(r)        = Update[s,r]}


and Trans(z) =
      z?{stat(r)    = r!done | Trans[z],
         list(k,r)  = r!done | Trans[z],
         retr(k,r)  = r!done | Trans[z],
         rset(r)    = r!done | Trans[z],
         del(k,r)   = r!done | Trans[z],
         noop(r)    = r!done | Trans[z],
         quit(r)    = Update[z,r]}

and Update(z,r) = -- update state after transactions | r!bye
```

4.3 Comparison with uniform TyCO

In a preliminary work studying a possible implementation for non-uniform TyCO, Luís Lopes and the author propose a *non-uniform abstract machine* [RL99], and discuss on the advantages of using non-uniform objects. A brief overview follows.

4.3.1 A simple abstract machine

The machine grows from a similar specification for the uniform case [LSV99]. The changes are very localised and essentially regard the structure of channels. In fact, in the uniform case the abstract machine preserves the invariant that, at any moment in a computation, the queue associated to a name is either empty or has only messages, or has only objects. In this case, to try to reduce a message, we just need to find out if the queue holds objects. Moreover, since record types are uniform, we know that any object in the queue will service the message. In the non-uniform case, due to the new notion of error, the invariant no longer holds. We can guarantee, however, that there are no redexes in each queue. In terms of the implementation, we need to keep track of the objects and messages in the queue, or more specifically, their labels. When a message arrives at the queue one must check whether there is an object in the queue that can service the message. A similar situation

Example	Non-uniform	Uniform
Buffer	30	51
ATM	303	332
FTP	183	190
POP3	158	142

Table 4.1: Assembler code size in words of the examples.

occurs with messages in a queue. One must keep track of the label maps for the methods of the objects in the queue. When a message arrives, one must match its label with the maps for the objects to find one that may service the request.

The fact that adapting the abstract machine for uniform TyCO to the non-uniform scheme takes such a small and localised effort gives us some hope that we may profit from the current implementation of TyCO, while keeping it efficient and compact. Nevertheless, an efficient implementation of channels is crucial, and this is a matter of finding the adequate data structure such that searching in the queue would not be a too costly operation.

4.3.2 Code size

Programming using non-uniform objects can frequently result in much shorter code, both in the source programs as well as in the final byte code or other executable format. This is due to the fact that, with non-uniform types, the implementation of an object is typically divided into several instances, each representing distinct aspects of the same object, reflecting the dynamic changes in the object's internal state. With uniform TyCO programming such objects require some extra work, and usually involve:

1. using distinct channels for each of the object's interfaces, since the type system does not allow objects with distinct interfaces at the same channel (e.g. the ATM example);
2. alternatively, the object may present at all times the same interface, independent of its internal state. It is up to the programmer then to decide what to do when a method is invoked and the object is not in a suitable state (e.g. the Buffer example in Section 4.1). This typically makes the programs cumbersome to read and, sometimes, substantially longer than a corresponding non-uniform implementation.

Table 4.1 presents the code size for the examples of Section 4.2. In all cases, but that of POP3, the code size is smaller in the non-uniform version. In the POP3 is larger, because the definitions have more parameters. Nevertheless, it is necessary to study this matter more thoroughly.

Even in cases where the difference in code size is not significant, there are two important advantages in non-uniform objects:

1. we use a single channel for all the object's dynamic instances. In the uniform case, a channel must be used for each distinct dynamic instance of an object, despite they are really different aspects of the same object. Thus we can use substantially less channel space in the heap for the non-uniform case;
2. since each dynamic aspect of an object typically features a small subset of the object's total interface and also of the object's free variables, the heap space that we allocate for each of these objects are smaller than those in the uniform case. Again we can optimise heap usage.

Therefore, the main advantage from an implementation point of view is the *optimisation of heap usage*.

An example of optimisation: linear channels [KPT96], namely those of multiplicity 1, can be highly optimised if using non-uniform objects. For example, assume you have the following chain of synchronisations on names a, b, and c:

$$\nu a\, x!m[va] \mid a?\{l(y) = \nu b\, y!m[ub] \mid b?\{k(z) = \nu c\, z!m[wc] \mid c?\{j(w) = ...\}\}\}\,.$$

Each successive synchronisation is guarded by the previous one. This means that once a, a linear channel, is used for synchronisation, it can be discarded, and theoretically, it could be used again in the next synchronisation, instead of a new channel b (and c for that matter) being allocated:

$$\nu a\, x!m[va] \mid a?\{l(y) = y!m[va] \mid a?\{k(z) = z!m[wa] \mid a?\{j(w) = ...\}\}\}$$

However, the uniform type system would prevent such usage, since the types for the objects do not match (labels l, k and j prevent it). With non-uniform types, this optimisation poses no problems whatsoever. We can perform all nested synchronisations with a single channel (here a) that behaves in a *quasi-linear* fashion [Kob99]. This optimisation should greatly improve performance, mostly by reducing the amount of heap required for an application. Synchronisation on a linear channel is a pervasive operation in most concurrent programs. It is important to observe that the type system defined in Chapter 14 supports this optimisations, albeit being very different from those referred above ([KPT96],[Kob99]), and pursuing a different, more general, purpose.

It is also important to notice that the type information allows that the runtime system may free the name after the last synchronisation, optimising garbage collection.

4.4 IMPLEMENTATION

To be able of really test these ideas, it is necessary to implement non-uniform TyCO. We are working on the definition of an abstract machine, namely on proving its correctness with respect to the operational semantics. Nevertheless, to fully explore non-uniformity, a type inference algorithm must be available and coupled with the compiler. That is, however, a more complex task, requiring a definitive answer to the problem of the decidability and computability of such an algorithm. In the conclusions of the thesis we discuss this topic in more detail.

Chapter 5

A dynamic notion of communication errors

In an object-oriented language, a communication error arises when a message targeted to some object requests a method that: (1) does not exist in the object; (2) is available in the object, but with a wrong number of arguments. A communication error of this kind arising at run-time is often designated as a '*message-not-understood*' error, the two reasons above clearly explaining why.

To deal with non-uniform service availability of concurrent objects, this static notion of error is unsuitable, since messages requesting methods that may become enabled at some time in the future should be considered valid. Thus, a dynamic notion of communication error should be able to detect messages that are never accepted, either because the requested method does not exist at all in the target object, or because, in spite of the object possessing such a method, this is not currently available, and furthermore, the object is, or will become, blocked, and then it cannot change its state to accept the request.

Therefore, we propose a looser understanding of what is a process with a communication error. The two situations mentioned above, being responsible for message-not-understood errors, are no longer considered as such if the request may be accepted by the object at some time in the future (after changing its state).

Recall that a type system is sound if ensures that no well-typed process will attain a communication error. Therefore, we need to define what is an error-free process. In this chapter we gradually build this notion, providing the necessary terminology and concepts, and illustrating through examples. The chapter concludes with a discussion on what is achieved.

5.1 A dynamic notion of error

In TyCO, a (static) communication error is an object-message pair, such that message application is not defined (cf. Definition 3.2.3 in page 20). It is a typical 'message-not-understood' error, originated by one of the reasons referred before. An object-message pair sharing the same subject is called a *redex*. If message application is defined, then the redex may reduce, and we call it a *good* redex; otherwise, it is a *bad* one. The classical notion of error considers that a process may cause a run-time error if one of its derivatives contains a bad redex. Relaxing this condition leads to a dynamic notion of run-time errors.

Notation and terminology. In this chapter, m denotes only input actions.[1] Then, $\Longrightarrow$ denotes $\xrightarrow{\tau}{}^*$, $\stackrel{m}{\Longrightarrow}$ denotes $\Longrightarrow\xrightarrow{m}\Longrightarrow$, and if $\tilde{m}$ is $m_1 \cdots m_n$ then $\stackrel{\tilde{m}}{\Longrightarrow}$ denotes the relation $\stackrel{m_1}{\Longrightarrow} \cdots \stackrel{m_n}{\Longrightarrow}$. The process Q such that $P \stackrel{\tilde{m}}{\Longrightarrow} Q$ is a *derivative* of P. Moreover, 'object' mean both persistent, $(\mathsf{rec}\, X(\vec{x}).a?M)[\vec{v}]$, and ephemeral, $a?M$, and let 'parallel composition of P and Q' mean both $P \mid Q$ and $Q \mid P$.

Definition 5.1.1 (Redex).

1. *A* redex *is the parallel composition of an object and a message with the same subject.*
2. *We say that the object and the message* participate *in the redex.*
3. *The redex is* good *if message application is defined; otherwise, it is* bad.
4. *A process P* has a redex *R, if the redex R is a subterm of P not within a method.*

Since an object's location is not unique, there may be several redexes for the same message. Making use of the notion of distributed object, we define distributed redexes. To stress the identity of the object and of the labelled value involved in the redex, we call them $al\tilde{v}$-redexes, where a is the subject of the object in the redex and $l[\tilde{v}]$ is the labelled value of the message in the redex.

Definition 5.1.2 (Distributed redex).

1. *The parallel composition of a distributed object $P \equiv \Pi_{i\in I}\, a?M_i | \Pi_{j\in J}\, (\mathsf{rec}\, X(\vec{x}).a?M_j)[\vec{v}_j]$ and of a message $a!l[\tilde{v}]$ is an* $al\tilde{v}$-redex, *which is* good *if, for some $k \in I \cup J$, message application is defined, and it is* bad *otherwise.*
2. *A process $P \equiv \nu\tilde{x}\, \Pi_{i\in I}\, a?M_i \mid \Pi_{j\in J}\, (\mathsf{rec}\, X(\vec{x}).a?M_j)[\vec{v}_j] \mid a!l[\tilde{v}] \mid Q$ where there is no input action m with subject a such that $Q \stackrel{m}{\Longrightarrow}$,* has an $al\tilde{v}$-redex.
3. *An $al\tilde{v}$-redex is* persistent *in a process P if all its derivatives contain such a redex.*

[1]They suffice due to the locality condition, as no context may create objects located at extruded names.

One cannot ensure that the object or the message participating in a persistent redex are always the same. In a persistent bad $al\tilde{v}$-redex, however, the message is indeed always the same, although the objects involved may change, and message application is never defined. A process is an error if, regardless of what kind of interactions it performs with the environment, at some point it may attain a derivative which has a persistent bad $al\tilde{v}$-redex. To be able to characterise this independence condition, we need to capture some causality between an environment and a redex emerging from a process in that environment.

Definition 5.1.3 (Redex generation).

1. *A non-empty sequence* $\widetilde{m}$ makes an $al\tilde{v}$-redex emerge from P by substituting $x_0\tilde{x}$*, if*
   (a) *P has a subterm $x_0!l[\tilde{x}]$ within a method $l(\tilde{w}) = Q$ with $\{x_0\tilde{x}\} \cap \{\widetilde{w}\} \neq \emptyset$, and*
   (b) *$P \stackrel{\widetilde{m}}{\Longrightarrow} \nu\tilde{z}\, Q[a\tilde{v}/x_0\tilde{x}][\tilde{u}/\widetilde{w}'] \mid R \Longrightarrow P'$, for some $\tilde{u}$ and $\{\widetilde{w}'\} = \{\widetilde{w}\} \setminus \{x_0\tilde{x}\}$, and P' has an $al\tilde{v}$-redex.*
2. *A sequence* $\widetilde{m}$ generates an $al\tilde{v}$-redex in P*, if there is $x_0\tilde{x}$ such that*
   (a) *$\widetilde{m}$ makes an $al\tilde{v}$-redex emerge from P by substituting $x_0\tilde{x}$, and*
   (b) *not all $\tilde{n}$ make an $al\tilde{v}$-redex emerge from P by substituting $x_0\tilde{x}$.*

5.1.1 Intuition

Error-free processes do not generate bad $al\tilde{v}$-redexes that persist throughout a computation, meaning that, even if a derivative of the process has a bad $al\tilde{v}$-redex, it is not persistent — the message is eventually consumed, or the object is ephemeral. Thus, an occasional bad redex is not enough to make the process an error, if further computation can consume (at least) one of the components of the bad redex. Therefore, messages that may be accepted at some time in the future do not cause errors. In short, errors are *persistent bad $al\tilde{v}$-redexes*. Recall that the environment can only send messages, i.e. does not provide objects located at the names of the processes under consideration.

We characterise the different kinds of situations leading to errors with some examples.

Example 5.1.4. Consider the one-place buffer in Section 4.1, page 26.

1. Process $\mathsf{Empty}[b] \mid b!\mathit{think}$ is an error, as the object b will never have the method *think*.
2. Processes $R \stackrel{\text{def}}{=} \mathsf{Empty}[b] \mid b!\mathit{read}[r]$ and $S \stackrel{\text{def}}{=} \mathsf{Empty}[b] \mid b!\mathit{write}[u] \mid b!\mathit{write}[v]$ are not errors, since the bad redexes can disappear if the environment provides the right messages. However, processes $\nu b\, R$ and $\nu b\, S$ are erroneous, since in both processes the object b is blocked (as the scope of the object's name is restricted). Hence, the bad redexes become persistent (cf. an example by Boudol [Bou97]).

   Notice that $\nu b\,(R \mid a!l[b])$ is not an error, since the name b can be extruded, so the bad redex is not necessarily persistent.

In conclusion, communication errors are processes containing a message that can never be accepted by a persistent object. We can distinguish two cases:

1. *message-never-understood*: the object does not, and will never, have the method requested by the message (cf. the first example above);
2. *blocking*: the object has the method requested by the message, but not in its present interface, and it can never reach the state where it could accept the message (cf. the second example above).

Only persistent objects may participate in persistent bad redexes. However, the key element of the bad redex is the labelled value of a message participating in it. It is this labelled value that causes the error, and we want to keep track of it.

EXAMPLE 5.1.5.

1. Process $a?\{l = a!m\}$ is not an error, as the object a is ephemeral, and thus, no persistent bad redex can possibly arise. However, the corresponding persistent object $(\textsf{rec}\, X(a;a).a?\{l{=}\ a!m \mid X[a;a]\})[a;a]$ is an error, since if it receives an l-message it produces a persistent bad am-redex.
2. Process $(\textsf{rec}\, X(x;y).x?\{l{=}\ y!l \mid x?\{m{=}\ y!m \mid X[x;y]\}\})[a;a] \mid a!l \mid a!m$ is not an error, although there is always a bad redex, but with different messages (it alternates between label l and m). The bad redex is recurring, but not persistent.
3. Process $(\textsf{rec}\, X(xy;xy).x?\{l{=}\ X[xy;xy] \mid x!l \mid y!n \mid y?n\})[ab;ab] \mid a!l \mid b!m$ is an error, even though the object participating in the bad bm-redex is always a different one.

We do not want to reject processes that compute erroneously due only to the incorrect behaviour of the environment — which can always interact incorrectly with a process, and thus produce errors — since many processes with free input-names would become errors. The environment can directly provoke errors, by sending messages with subjects and/or objects that will be involved in the bad redex, or indirectly, by sending messages that make the bad redex emerge, although the names involved in the redex do not come from the environment. The examples below illustrate these two different situations.

EXAMPLE 5.1.6.

1. Process $Q \stackrel{\text{def}}{=} \textsf{Empty}[b] \mid (\textsf{rec}\, X(c;).c?\{l(x){=}\ x!\mathit{think}\})[c;]$ is not an error, although there are interactions leading to errors. The subject of the bad redex may be an object of a message from the environment, as in

   $Q \xrightarrow{c?l[b]} \textsf{Empty}[b] \mid b!\mathit{think} \mid (\textsf{rec}\, X(c;).c?\{l(x){=}\ x!\mathit{think}\})[c;]$.

   Likewise, the environment can send a message with a subject that will be the subject in the bad redex, like in $Q \xrightarrow{c?l[c]} \textsf{Empty}[b] \mid (\textsf{rec}\, X(c;).c?\{l(x){=}\ x!\mathit{think}\})[c;] \mid c!\mathit{think}$, or even a message with a object that will be an object in the bad redex.

2. None of the names in the bad $al\tilde{v}$-redex is a subject or an object of some message from the environment.

   $P \stackrel{\text{def}}{=} (\textsf{rec}\, X(x;).x?\{l = X[x;]\})[a;] \mid b?\{l(x) = \nu v\, x!l[v] \mid v?\{l(x) = x!l\} \mid c?\{l(x) = x!l[a]\}$ is not an error, although it has derivatives which are errors, depending on the requests from the environment.

   So, $P \xrightarrow{b?l[c]} \xrightarrow{\tau} \xrightarrow{\tau} \xrightarrow{\tau} (\textsf{rec}\, X(x).x?\{l = X[x;]\})[a]$, which is error-free, but $P \xrightarrow{c?l[b]} \xrightarrow{\tau} \xrightarrow{\tau} \nu v\, (\textsf{rec}\, X(x).x?\{l = X[x;]\})[a] \mid v?\{l(x) = x!l\} \mid a!l[v]$, which is an error.

5.1.2 The definition of error-free processes

The main safety result of a type system is the absence of run-time errors for well-typed processes. The notion of run-time error-free process collects all the principles stated and explained above.

Definition 5.1.7 (Error-Free Process). *The process P is* error-free *if, for all $al\tilde{v}$,*

$$\forall_{Q,\tilde{m}}\, P \stackrel{\tilde{m}}{\Longrightarrow} Q \text{ where } Q \text{ has a bad } al\tilde{v}\text{-redex not generated by } \tilde{m} \text{ implies}$$
$$\exists_{R,\tilde{n}}\, Q \stackrel{\tilde{n}}{\Longrightarrow} R \text{ where } R \text{ does not have a bad } al\tilde{v}\text{-redex.}$$

5.2 Remarks

Some comments on this notion are appropriate now. Observe that the definition is of *error-free processes*, not of errors, since this is the notion required to prove the soundness of the type system. Moreover, it turns out to be much simpler to define than its negation.

5.2.1 Limits of the notion

This notion of error remains limited by an expected problem: it is undecidable. There are error-processes with infinite runs where the bad redex never appears, as there is no fairness assumption in the calculus. Actually, for any computational mechanism with the same power than a Turing machine, verifying if a program will, at run-time, produce an error, is a problem equivalent to the halting problem, or to the problem of deciding whether a lambda term has a normal form. This has been recently proved in the context of the π-calculus [VR99].

An approach commonly used to overcome this difficulty is that of type systems, which permit the detection of all the potential errors, i.e. it is sound. Due to the undecidability problem, this approach cannot possibly be complete. Hence, it fails to type-check some processes that do not conform to the notion of error.

5.2.2 An alternative

We do not consider as errors, neither messages without target, nor blocked objects, by themselves, as they are not communication errors. Processes like $\nu a\, a!l$ or $\nu a\, a?M$ are deadlocked, and should be garbage collected. Detecting processes that may attain one of the previous situations is a more delicate matter than to ensure program safety. Furthermore, the guarantee that all messages are treated, or that a process do not contain blocked objects or is not deadlocked, correspond to ensure liveness properties. A more interesting property than simple 'orphan-free' is to ensure that every request of a method is attended, this property being a merged of the two previously referred.

Defining processes without 'orphan messages' is simple. Actually, the definition of error-free processes detects some 'orphans', thus, such a definition would generalise ours. In consequence, the type system would be more complicated, although the changes would be localised.

5.2.3 A related notion

This notion of communication error-free process is, to our knowledge, original. It seems quite natural to have a liberal approach to potential errors in the context of non-uniform concurrent objects. We claim that it allows a more flexible and behaviour-oriented style of programming and, moreover, detects some deadlocks.

In the context of the lazy λ-calculus [Abr90], Dami proposes a liberal approach to potential errors [Dam97], which is similar to our notion. He argues that the common notion of erroneous term is over-restrictive: some programs, in spite of having error terms in them, do not actually attain a run-time error when executed. Since there is a family of programming languages based on the lazy λ-calculus, Dami proposes a lazy approach to errors: a term is considered erroneous if and only if it always generates an error after a finite number of interactions with its context.

Part III

The algebra of behavioural types

Chapter 6

Motivation

The purpose of this part is to study the semantic foundations of types for non-uniform concurrent objects. In particular, we discuss a notion of equivalence of types, and develop an axiomatic system for it.

In a name-passing calculus such as TyCO, processes denote the behaviour of a community of interacting objects, where each object has a location identified by a name [Vas94a]. The processes determine an assignment of types to names that reflects a discipline for communication. The usual types-as-records paradigm gives each name a static type that contains information about all the methods of the object, regardless of whether they are enabled or not [VT93]. To cope with non-uniform service availability [Nie95], the types should reflect a dependency between the interface of an object and its internal state, conveying information about dynamic properties of objects. The aim is to build type systems capable of ensuring not only the usual safety properties like subject-reduction, but also more complex properties, like the absence of some deadlocks.

Herein we develop the *Algebra of Behavioural Types*, ABT, where a type is basically a collection of enabled methods (an interface); such a type is dynamic in the sense that the execution of a method can change the type. Therefore, the type of an object is a partial representation of its behaviour modelled as a labelled transition system. We assume that objects communicate via asynchronous message-passing; nevertheless, types, as defined here, essentially correspond to a notion of object behaviour as it would be perceived by an internal observer located within an object (the object's private "gnome"). This observer can see methods being invoked and can detect whether the object is blocked, even though its methods may be internally enabled. Hence, this notion of behaviour is synchronous, as the gnome can detect refusals of methods. The action of unblocking an object, denoted by υ, cannot be directly observed by the gnome because it corresponds to an invocation of a method in another object. Thus, this action is similar to CCS's τ in that it is hidden, but it is external rather than internal [Mil89a]. Therefore, we reach a set of laws different from Milner's τ-laws.

ABT is similar to the Basic Parallel Processes, BPP, a fragment of CCS proposed by Christensen where communication is not present (parallel composition is simply a merge) [Chr93]. There are three main differences:

1. sums are prefixed;
2. mixed sums are not allowed; and
3. the silent action is external rather than internal.

Apart from this, the notion of equivalence has also an intuition different from the one for bisimulations in CCS, since the observer is internal, rather than external. Therefore, instead of using BPP, we design our own process algebra.

We build ABT gradually. The next chapter presents finite types built with a non-deterministic labelled sum and a blocking operator; defines the operational semantics and an equivalence notion; and provides a complete axiomatisation for that equivalence. In Chapter 8 we add a parallel composition operator, this operator being a merge of processes (i.e. without communication), and extend the previous axiomatic system with two expansion laws — consequence of the absence of mixed sums — and a saturation law. Since the normal forms include parallel compositions, the proof of completeness of the axiomatic system is not standard, and the result depends on a new inference rule that we add to the proof system of equational logic. The rule does not seem to be derivable. However, we conjecture that the axiomatic system is complete without this inference rule, but the result is not yet established, as it depends on a conjecture . We discuss the problem in the appendix. Finally, Chapter 9 presents the full algebra, with dynamic types obtained by a recursive constructor. The axiomatic system contains three more laws to deal with recursion. In Chapter 10 we discuss the completeness of the axiomatic system, both for image-finite and image-infinite types. This part closes with a chapter on comparisons with related work and on some directions for future research, namely a modal logic and a notion of subtyping. The results presented herein are clearly preliminary.

Chapter 7

Non-deterministic finite types

We start by presenting an algebra of *non-deterministic sequential finite types*. The basic term is an *object type*, a finite collection of methods denoted by a labelled sum that stand for an interface of an object. As we allow the same label to appear more than once (possibly with different continuations under the prefix), the sum is non-deterministic. This fact makes possible the definition of an expansion law later on when we introduce a parallel composition operator.

We construct unavailable, i.e. *blocked*, object types by means of a blocking operator — denoted by υ — that prefixes object types. A sum of blocked object types is a sum of possible types of an object, depending on the state of the system. Hence, the silent transition is labelled with υ and corresponds to the release of the blocked type; it is an *inter-object choice* that makes available one of the types in the sum. Thus, it should be interpreted as an action that is external to the object. Obviously, we want to distinguish an object that is enabled and offers a certain collection of methods from one that it is blocked.

We did not find in the realm of process algebra any equivalence notion build with this intuition. Therefore, we develop a notion of type equivalence accordingly to the requirements explained above.

The intended meaning of a labelled sum and of a blocked sum clarify that it does not make sense to allow mixed sums. This fact leads to axiomatic systems that are not standard and require new proof techniques. This is probably one of the most interesting aspects of ABT, which is starting now to be developed.

7.1 SYNTAX

Take the countable set of *method names* l, m, possibly subscripted, assumed in Definition 3.1.1.

DEFINITION 7.1.1 (NON-DETERMINISTIC SEQUENTIAL FINITE TYPES).
The following grammar defines the set $\mathcal{T}_{sf}$ *of* sequential finite types.

$$\alpha ::= \sum_{i\in I} l_i(\widetilde{\alpha}_i).\alpha_i \;\;|\;\; \sum_{i\in I} \upsilon.\alpha_i$$

where I *is a finite, possibly empty, indexing set, and each* $\widetilde{\alpha}_i$ *is a finite sequence of types.*

A term of the form $l(\widetilde{\alpha}).\alpha$ is a *method type*. The label l in the prefix stands for the name of a method possessing parameters of type $\widetilde{\alpha}$; the type α under the prefix prescribes the behaviour of the object after the execution of the method l with parameters of type $\widetilde{\alpha}$. A term of the form $\upsilon.l(\widetilde{\alpha}).\alpha$ is a *blocked method type*, the type of an unavailable method type $l(\widetilde{\alpha}).\alpha$.

The only type composition operator of the algebra is the sum, '$\sum$', which has two uses:

1. gathers together several method types to form the type of an object that offers the corresponding collection of methods: the *labelled sum* $\sum_{i\in I} l_i(\widetilde{\alpha}_i).\alpha_i$;

2. associates several blocked types in the *blocked sum* $\sum_{i\in I} \upsilon.\alpha_i$; after being released, the object behaves according to one of the types α_i.

NOTATION. We write $\alpha \equiv \beta$ when the types α and β are *syntactically identical*. Consider also the following abbreviations:

1. $\mathbf{0}$ denotes the empty type (sum with empty indexing set), we omit the sum symbol if the indexing set is singular, and we use the plus ('+') to denote binary sums of types, which we assume associative;

2. $l(\widetilde{\alpha})$ denotes $l(\widetilde{\alpha}).\mathbf{0}$, and l denotes $l()$.

Henceforth we call *blocked types* the blocked sums.

DEFINITION 7.1.2 (INTERFACE OF A TYPE).
In a labelled sum $\alpha \equiv \sum_{i\in I} l_i(\widetilde{\alpha}_i).\alpha_i$ *the set* $\{l_i(\widetilde{\alpha}_i) \mid i \in I\}$ *is the* interface *of the object type, denoted by* $\mathrm{int}(\alpha)$*. The interface of a blocked type is empty.*

7.2 Operational semantics

A labelled transition relation on types defines the structural operational semantics for non-deterministic sequential finite types.

Definition 7.2.1 (Prefixes).
The following grammar defines the set of prefixes*:*

$$\pi ::= \upsilon \ \mid \ l(\widetilde{\alpha})$$

The prefix υ denotes a silent transition that releases a blocked object; a prefix like $l(\widetilde{\alpha})$ denotes a transition corresponding to the invocation of method l with actual parameters of types $\widetilde{\alpha}$. We write $\sum_{i\in I}\pi_i.\alpha_i$ to refer to an arbitrary sum, either with prefixes $l_i(\widetilde{\alpha}_i)$ — *labelled* — or with prefix υ — *blocked.*

Definition 7.2.2 (Labelled transition relation).
The following rule inductively defines a labelled transition relation *on* $\mathcal{T}_{sf}$*.*

$$\textsc{Act} \quad \sum_{i\in I}\pi_i.\alpha_i \xrightarrow{\pi_j} \alpha_j \quad (j \in I)$$

Act is in fact an axiom-schema that captures two cases:

1. the basic transition $l(\widetilde{\alpha})$ — *execution of a method* — is the invocation of a method with name l and parameters of types $\widetilde{\alpha}$, yielding the type α of the object in the method body;
2. a silent transition υ — *unblocking* — releases a blocked type.

Terminology. Some terminology regarding the transition relation will make the proofs clearer.

1. If $\alpha \xrightarrow{l(\widetilde{\alpha})} \alpha'$ or $\alpha \xrightarrow{\upsilon} \alpha'$ then α' is a *derivative* of α. In the first case it is an l-derivative and the second an υ-derivative.
2. Types of the form $\sum_{i\in I}l_i(\widetilde{\alpha}_i).\alpha_i$ are *unblocked*, and types of the form $\sum_{i\in I}\upsilon.\alpha_i$ are *blocked.* If $I \neq \emptyset$, $\sum_{i\in I}l_i(\widetilde{\alpha}_i).\alpha_i$ is *strictly unblocked*, and $\sum_{i\in I}\upsilon.\alpha_i$ is *strictly blocked.*

Notation. Let $\Longrightarrow$ denote $\xrightarrow{\upsilon}{}^{*}$ and $\overset{\upsilon}{\Longrightarrow}$ denote $\xrightarrow{\upsilon}{}^{+}$.

7.3 EQUIVALENCE NOTION

We want two types to be equivalent if they offer the same methods — have the same interface — and if, after each transition, they continue to be equivalent, in a bisimulation style. Furthermore, from the point of view of each type, transitions of other types can be regarded as hidden transitions, which would suggest weak bisimulation as the right notion of equivalence for our types, with υ playing the role of Milner's τ, but representing external interaction rather than internal. However, we want types to distinguish an object that immediately makes available a method from another that makes it available only after being unblocked (by some object). This is because, although υ is supposed to be unobservable, we assume that an internal observer — the object's gnome — can detect that the object is blocked. Hence, we would expect $\upsilon.l$ to be different from l, since all the internal observer can see is that the object is blocked, and after being released it can eventually execute the method l. This discards weak bisimulation as a candidate for type equivalence. Furthermore, we want $\upsilon.l$ and $\upsilon.\upsilon.l$ to be equivalent, because the number of unblockings cannot be counted from within the object as they correspond to transitions on other objects, thus discarding strong bisimulation [Mil89a] and progressing bisimulation [MS92]. We also want to distinguish $l.\upsilon.m$ from $l.m$ on the grounds that, for the latter, a blocking after l cannot be observed, and thus observational congruence [Mil89a] and rooted bisimulation [BBK87] are unsuitable. Also, notice that all the above mentioned equivalences, with the exception of weak bisimulation, are finer than necessary, because they are congruences with respect to binary sums, as in CCS [Mil89a], whereas in this thesis we stick to prefixed sums.

These considerations lead to the choice of a notion of equivalence that we call *label-strong bisimulation*, or *lsb*. It is a higher-order strong bisimulation on labels and a weak bisimulation on unblockings.

Hence, we require that if α and β are bisimilar then:

1. if α offers a particular method, then also β offers that method, and the parameters and the bodies of the methods are pairwise bisimilar;
2. if α offers a hidden transition, then β can offer zero or more hidden transitions.

DEFINITION 7.3.1 (BISIMILARITY ON TYPES).

1. *A symmetric binary relation* $R \subseteq \mathcal{T}_{sf} \times \mathcal{T}_{sf}$ *is a* label-strong bisimulation*, or simply a bisimulation, if, whenever* $\alpha \, R \, \beta$*:*
   (a) $\alpha \xrightarrow{l(\widetilde{\alpha})} \alpha'$ *implies* $\exists \beta', \widetilde{\beta}$ $(\beta \xrightarrow{l(\widetilde{\beta})} \beta'$ *and* $\alpha'\widetilde{\alpha} \, R \, \beta'\widetilde{\beta})^1$*;*
   (b) $\alpha \xrightarrow{\upsilon} \alpha'$ *implies* $\exists \beta'$ $(\beta \Longrightarrow \beta'$ *and* $\alpha' \, R \, \beta')$*;*
2. *Two types* α *and* β *are* label-strong bisimilar*, or simply bisimilar, and we write* $\alpha \approx \beta$*, if there is a label-strong bisimulation* R *such that* $\alpha \, R \, \beta$*.*

[1] Let $(\alpha_1 \cdots \alpha_n) \, R \, (\beta_1 \cdots \beta_n)$ denote $\alpha_1 \, R \, \beta_1, \ldots,$ and $\alpha_n \, R \, \beta_n$.

Thus, the intuition is: two types are *not bisimilar* if they have different interfaces or, after some matching transitions, their derivatives have different interfaces.

The usual properties of bisimilarities hold, namely *lsb* is an equivalence relation and a fixed point. Also, it is simple to give an alternative characterisation of *lsb*.

PROPOSITION 7.3.1 (EQUIVALENCE).
Label-strong bisimulation is an equivalence relation.

PROOF. One easily checks that it is reflexive, symmetric, and transitive. □

PROPOSITION 7.3.2 (FIXED POINT).
Types α and β are bisimilar, if, and only if,

1. $\alpha \xrightarrow{l(\widetilde{\alpha})} \alpha'$ *implies* $\exists \beta', \widetilde{\beta}$ $(\beta \xrightarrow{l(\widetilde{\beta})} \beta'$ *and* $\alpha'\widetilde{\alpha} \approx \beta'\widetilde{\beta})$;
2. $\alpha \xrightarrow{\upsilon} \alpha'$ *implies* $\exists \beta'$ $(\beta \Longrightarrow \beta'$ *and* $\alpha' \approx \beta')$.

PROOF. The proof is standard (cf. [Mil89a], Proposition 4.16). □

PROPOSITION 7.3.3 (LABEL-STRONG BISIMILARITY).
Types α and β are bisimilar, if, and only if,

1. $\alpha \xrightarrow{l(\widetilde{\alpha})} \alpha'$ *implies* $\exists \beta', \widetilde{\beta}$ $(\beta \xrightarrow{l(\widetilde{\beta})} \beta'$ *and* $\alpha'\widetilde{\alpha} \approx \beta'\widetilde{\beta})$;
2. $\alpha \Longrightarrow \alpha'$ *implies* $\exists \beta'$ $(\beta \Longrightarrow \beta'$ *and* $\alpha' \approx \beta')$.

PROOF. Notice that $\alpha \Longrightarrow \alpha'$ means $\alpha \xrightarrow{\upsilon}^{n} \alpha'$, for some natural number n. The proof is by induction on n. □

The sum of method types and the sum of blocked types preserve bisimilarity. This result is simple to verify, since both sums are guarded.

PROPOSITION 7.3.4 (CONGRUENCE).
Let $\widetilde{\alpha}_i \approx \widetilde{\beta}_i$ and $\alpha_i \approx \beta_i$ for all i in some indexing set I.

1. $\sum_{i \in I} l_i(\widetilde{\alpha}_i).\alpha_i \approx \sum_{i \in I} l_i(\widetilde{\beta}_i).\beta_i$, *and*
2. $\sum_{i \in I} \upsilon.\alpha_i \approx \sum_{i \in I} \upsilon.\beta_i$.

Briefly, *lsb* is a (higher-order) congruence relation with respect to prefixing and summation.

PROOF. Proving that labelled sums and blocked sums preserve *lsb* is a direct application of Definition 7.3.1, and of the fixed point property of *lsb* (Proposition 7.3.2). □

7.4 ALGEBRAIC CHARACTERISATION

We present an axiomatisation of the equivalence notion and show that it is sound and complete. The proof technique for completeness is standard: a definition of a normal form for the types; a lemma ensuring that for all types there exists an equivalent term in normal form; and finally the completeness theorem says that for all pairs of equivalent normal forms there exists a derivation of their equality using the rules of the axiomatic system.

PROP/DEFINITION 7.4.1 (AXIOMATISATION).
The following equivalences inductively define the axiomatic system $\mathcal{A}_{sf}$.

Commutativity : *for any permutation* $\sigma : I \to I$ *we have* $\sum_{i\in I} \pi_i.\alpha_i \approx \sum_{i\in I} \pi_{\sigma(i)}.\alpha_{\sigma(i)}$;
Idempotence : $\pi.\alpha + \pi.\alpha + \beta \approx \pi.\alpha + \beta$;
υ**-law** : $\upsilon.\sum_{i\in I} \upsilon.\alpha_i \approx \sum_{i\in I} \upsilon.\alpha_i$.

PROOF. It is straightforward to build the respective bisimulations. □

THEOREM 7.4.1 (SOUNDNESS OF $\mathcal{A}_{sf}$).
If $\vdash \alpha = \beta$ *then* $\alpha \approx \beta$.

PROOF. Follows from Proposition 7.3.4 and of Prop/Definition 7.4.1. □

REMARK.

1. We write $\vdash \alpha = \beta$ when we can prove $\alpha \approx \beta$ using the laws above and the usual rules of equational logic. The proof of completeness uses two derived rules.

$$\text{If } \forall_{i\in I}\exists_{j\in J} \vdash \alpha_i = \beta_j \text{ and } \forall_{j\in J}\exists_{i\in I} \vdash \alpha_i = \beta_j\text{, then } \vdash \sum_{i\in I} \upsilon.\alpha_i = \sum_{j\in J} \upsilon.\beta_j \quad (7.1)$$

$$\text{If } \forall_{i\in I}\exists_{j\in J} \vdash \alpha_i = \beta_j \text{ and } \forall_{j\in J}\exists_{i\in I} \vdash \alpha_i = \beta_j \text{ and } \forall_{i\in I}\exists_{j\in J}\, (l_i \equiv m_j \text{ and } \vdash \widetilde{\alpha}_i = \widetilde{\beta}_j)$$
$$\text{and } \forall_{j\in J}\exists_{i\in I}\, (l_i \equiv m_j \text{ and } \vdash \widetilde{\alpha}_i = \widetilde{\beta}_j)\text{, then } \vdash \sum_{i\in I} l_i(\widetilde{\alpha}_i).\alpha_i = \sum_{j\in J} m_j(\widetilde{\beta}_j).\beta_j \quad (7.2)$$

The soundness of these rule is a consequence of Proposition 7.3.4, but it results also from the fact that they are derived rules (the proof is simple). Therefore, the proof system without them is still complete.

2. An interesting instance of the υ-law is $\alpha \approx \mathbf{0}$ if α is a blocked sum with all its derivatives being also blocked sums (i.e. α is a tree with all branches labelled by υ).

3. The υ-law corresponds to an instance of the CCS's τ-law $\alpha.\tau.P = \alpha.P$ where $\alpha \equiv \tau$.

   One can easily recognise particular instances of the remaining τ-laws of CCS that hold in this setting. For example, the following derivable laws are instances of the second and third τ-laws.

(a) $\upsilon.P + \upsilon.\upsilon.P = \upsilon.\upsilon.P$;
(b) $\upsilon.(\upsilon.P + \upsilon.Q) = \upsilon.(\upsilon.P + \upsilon.Q) + \upsilon.Q$;

The first clause is easy to prove, since $\upsilon.\upsilon.P = \upsilon.P$; to prove the second clause use first the υ-law, then idempotence, and then again the υ-law.

However, the τ-laws do not hold in general in this setting; for instance, the third one does not hold, as the following counter-example shows:

$$l.(\upsilon.m + \upsilon.n) \not\approx l.(\upsilon.m + \upsilon.n) + l.n\,.$$

From the remarks it is easy to conclude that weak bisimulation, which is a congruence for prefixed sums, and the observation congruence, are coarser than *lsb*.

DEFINITION 7.4.2 (NORMAL FORMS OF SEQUENTIAL FINITE TYPES).

1. *A type α is* saturated *if $\alpha \stackrel{\upsilon}{\Longrightarrow} \alpha'$ implies $\alpha \stackrel{\upsilon}{\rightarrow} \alpha'$.*
2. *A type α is a* normal form *if it is saturated and, furthermore, one of the following conditions holds:*
   (a) $\alpha \equiv \mathbf{0}$;
   (b) $\alpha \equiv \sum_{i \in I} \upsilon.\alpha_i$ *and each α_i is a normal form;*
   (c) $\alpha \equiv \sum_{i \in I} l_i(\widetilde{\alpha}_i).\alpha_i$*, where each component of each sequence $\alpha_i\widetilde{\alpha}_i$ is a normal form.*

DEFINITION 7.4.3 (DEPTH OF A TYPE).
The following rules inductively define the depth of a type*.*

$$\begin{array}{rcl} \mathrm{depth}(\mathbf{0}) & = & 0, \\ \mathrm{depth}(\sum_{i \in I} \upsilon.\alpha_i) & = & 1 + \max\{\mathrm{depth}(\alpha_i) \mid i \in I\},\ \textit{and} \\ \mathrm{depth}(\sum_{i \in I} l_i(\widetilde{\alpha}_i).\alpha_i) & = & 1 + \max\{\mathrm{depth}(\widetilde{\alpha}_i) + \mathrm{depth}(\alpha_i) \mid i \in I\}, \end{array}$$

where $\mathrm{depth}(\widetilde{\alpha}) = \max\{\mathrm{depth}(\beta) \mid \beta \in \widetilde{\alpha}\}$.

We prove now that all types have equivalent normal forms, a crucial lemma to achieve completeness.

LEMMA 7.4.2 (NORMAL FORM LEMMA).
For all types α there exists a normal form α' such that $\vdash \alpha = \alpha'$, with $\mathrm{depth}(\alpha') \leq \mathrm{depth}(\alpha)$.

PROOF. By induction on the depth of α. The base case is when $\mathrm{depth}(\alpha) = 0$; so, $\alpha \equiv \mathbf{0}$, and it is a normal form by definition. Otherwise, if $\alpha \equiv \sum_{i \in I} \pi_i.\alpha_i$ then, by induction hypothesis, for each α_i there exists a normal form α'_i such that $\vdash \alpha_i = \alpha'_i$, with $\mathrm{depth}(\alpha'_i) \leq \mathrm{depth}(\alpha_i)$. The prefix can be of two forms:

1. $\pi_i \equiv l_i(\widetilde{\alpha}_i)$, for all i.

   Since the types $\widetilde{\alpha}_i$ also have normal forms $\widetilde{\alpha}'_i$, we conclude that $\vdash \alpha = \sum_{i \in I} l_i(\widetilde{\alpha}'_i).\alpha'_i$, as clearly, $\mathrm{depth}(\sum_{i \in I} l_i(\widetilde{\alpha}'_i).\alpha'_i) \leq \mathrm{depth}(\alpha)$.

2. $\pi_i \equiv \upsilon$, for all i.

   We have to guarantee saturation. For each i such that $\alpha'_i \xrightarrow{\upsilon}$, there is a $J_i \neq \emptyset$ such that $\alpha'_i \equiv \sum_{j \in J_i} \upsilon.\alpha_j$ and, for each $j \in J_i$ we have $\sum_{i \in I} \upsilon.\alpha'_i \xrightarrow{\upsilon.\upsilon} \alpha_j$. By idempotence and the υ-law, we have $\vdash \sum_{i \in I} \upsilon.\alpha'_i = \sum_{i \in I} \upsilon.\alpha'_i + \upsilon.\alpha_j$. We can add $\upsilon.\alpha_j$ to $\sum_{i \in I} \upsilon.\alpha'_i$ for all such α_j and thus obtain a normal form of α whose depth clearly equals that of $\sum_{i \in I} \upsilon.\alpha'_i$.

The proof is complete. □

We are now in a position to prove the main result of this chapter: the completeness of the axiomatic system $\mathcal{A}_{sf}$ with respect to the notion of equivalence.

THEOREM 7.4.3 (COMPLETENESS OF $\mathcal{A}_{sf}$).
If $\alpha \approx \beta$ then $\vdash \alpha = \beta$.

PROOF. By induction on the sum of the depths of the types α and β (assumed to be normal forms, following Lemma 7.4.2).

Base case: $\text{depth}(\alpha) = \text{depth}(\beta) = 0$;

Then, $\alpha \approx \mathbf{0} \approx \beta$, and it follows from the reflexivity law of the proof system that $\vdash \alpha = \beta$.

Induction step: there are to cases to be consider.

1. α is a labelled sum, and hence also β is a labelled sum, as $\alpha \approx \beta$.

   Thus, if $\alpha \xrightarrow{l(\widetilde{\alpha})} \alpha'$ then $\beta \xrightarrow{l(\widetilde{\beta})} \beta'$ and $\alpha'\widetilde{\alpha} \approx \beta'\widetilde{\beta}$; it follows by the induction hypothesis that $\vdash \alpha'\widetilde{\alpha} = \beta'\widetilde{\beta}$;

2. α is a blocked sum, and hence also β is a blocked sum.

   Thus, if $\alpha \xrightarrow{\upsilon} \alpha'$ then $\beta \xrightarrow{\upsilon} \beta'$, and again we conclude that $\vdash \alpha' = \beta'$.

The result follows using the derived inference rules, 7.2 in the first case, and 7.1 in the second. □

Chapter 8

Concurrent finite types

Now we extend the algebra of non-deterministic sequential finite types with concurrent types, defining a parallel composition operator. Since the algebra does not have communication, this operator is simply a merge of types (cf. the parallel composition operator of BPP).

A type constructed with the parallel composition operator denotes the behaviour of a parallel composition of input processes with the same location. In the parallel composition of types each component is the type of an element of the parallel composition of input processes.

The axiomatic system includes now two expansion laws: the parallel composition of labelled sums is equivalent to a labelled sum; the parallel composition of blocked sums is equivalent to a blocked sum. However, the absence of mixed sums in the grammar of types prohibits a general expansion law. The main consequence of this fact is that normal forms include parallel compositions, and the standard proof technique to establish the completeness of the axiomatic system must be refined. In particular, we are forced to add a new inference rule to the proof system of equational logic, which we prove sound. The rule does not seem to be derivable, but we believe that it is possible that the axiomatic system without it is still complete.

An alternative proof of completeness uses only induction, but requires that the converse of the congruence for the parallel holds for normal forms. We attempt to prove this fact by transitional induction, but did not succeed (the appendix gives a detailed account of the reasoning). We conjecture that the result holds, and leave it as an open problem. Once proved, the conjecture can be used as a lemma in a simpler proof of the completeness of the axiomatic system for the notion of equivalence, system that would not require the new inference rule mentioned before.

8.1 Syntax

Take the set of method names assumed in the previous chapter.

DEFINITION 8.1.1 (CONCURRENT FINITE TYPES).
The following grammar defines the set $\mathcal{T}_f$ of concurrent finite types.

$$\alpha ::= \sum_{i \in I} l_i(\widetilde{\alpha}_i).\alpha_i \ \mid \ \sum_{i \in I} \upsilon.\alpha_i \ \mid \ \alpha \parallel \alpha$$

where I is a finite, possibly empty, indexing set, and each $\widetilde{\alpha}_i$ is a finite sequence of types.

The parallel composition operator, denoted by '$\parallel$', represents the existence of several objects located at the same name, an executing in parallel (interpreted as different copies of the same object, possibly in different states).

TERMINOLOGY. Unblocked types are now not only labelled sums, but also parallel compositions involving (at least) a labelled sum. Types that are not of the form $\sum_{i \in I} \upsilon.\alpha_i$ are *unblocked.* Furthermore, each α_i is *strictly blocked* if $I \neq \emptyset$, and a type is *strictly unblocked* if it has an l-derivative.[1]

Therefore, if α is an unblocked type, its interface is the union of the interfaces of the labelled sums that are involved in the parallel composition.

DEFINITION 8.1.2 (INTERFACE OF A TYPE).
The following rules inductively define the interface of a type.

$$\begin{array}{rcl} \mathrm{int}(\sum_{i \in I} \upsilon.\alpha_i) & = & \emptyset \\ \mathrm{int}(\sum_{i \in I} l_i(\widetilde{\alpha}_i).\alpha_i) & = & \{l_i(\widetilde{\alpha}_i) \mid i \in I\} \\ \mathrm{int}(\alpha_1 \parallel \alpha_2) & = & \mathrm{int}(\alpha_1) \cup \mathrm{int}(\alpha_2) \end{array}$$

NOTATION. Consider that the prefixes bind tighter than the parallel constructor, i.e., $l.m \parallel n$ is $(l.m) \parallel n$.

8.2 Operational semantics

Take the set of labels specified by Definition 7.2.1.

[1] Syntactic characterisations, e.g. via grammars, are easily defined.

Definition 8.2.1 (Labelled transition relation).
The axiom schema of Definition 7.2.2 together with the two rules below inductively define the labelled transition relation *of the algebra of concurrent finite types.*

$$\text{RPAR}\quad \frac{\alpha \xrightarrow{\pi} \alpha'}{\alpha \parallel \beta \xrightarrow{\pi} \alpha' \parallel \beta} \qquad\qquad \text{LPAR}\quad \frac{\alpha \xrightarrow{\pi} \alpha'}{\beta \parallel \alpha \xrightarrow{\pi} \beta \parallel \alpha'}$$

Proposition 8.2.1 (Congruence).
The parallel composition operator preserves the bisimilarity relation (cf. Definition 7.3.1).

Proof. It is easy to see that the relation $\{(\alpha \parallel \beta, \alpha' \parallel \beta') \mid \alpha \approx \alpha' \text{ and } \beta \approx \beta'\}$ is a bisimulation, and thus $\parallel$ preserves $\approx$. □

Characterisations of the types that are not bisimilar to $\mathbf{0}$ are useful. Syntactically, one can do it with a two-level grammar. Semantically, one uses the following lemma.

Lemma 8.2.2.
$\alpha \not\approx \mathbf{0}$*, if and only if,* $\exists_{\alpha', l(\tilde{\alpha})}\, \alpha \Longrightarrow \alpha' \xrightarrow{l(\tilde{\alpha})}$.

Proof. The 'if' direction follows easily by absurd, considering $\alpha' \approx \mathbf{0}$. The 'only-if' direction is by induction on the structure of α. □

The subsequent results make use of the notion of depth of a type.

Definition 8.2.2 (Depth of a type).
The rules of Definition 7.4.3, together with the rule below inductively define the depth of a type.

$$\text{depth}(\alpha \parallel \beta) = \text{depth}(\alpha) + \text{depth}(\beta)$$

The following result is a surprising consequence of the equivalence relation.

Proposition 8.2.3.
Let α be a labelled sum and γ be a blocked type, such that $\alpha \approx \beta \parallel \gamma$, for β a sum (either labelled or blocked). Then $\gamma \approx \mathbf{0}$.

Proof. If β is blocked then clearly $\alpha \approx \mathbf{0}$, hence $\gamma \approx \mathbf{0}$. For the remainder of this proof, let us assume that β is unblocked. The proof follows by induction on depth(α).

Base case: if depth$(\alpha) = 1$, then $\alpha \approx \beta$ and it follows easily that $\gamma \approx \mathbf{0}$.

For the induction step we assume $\gamma \not\approx \mathbf{0}$ and we derive a contradiction as follows. By Lemma 8.2.2 $\gamma \Longrightarrow \gamma' \xrightarrow{l(\tilde{\gamma})} \gamma''$ for some γ', l, $\tilde{\gamma}$, and γ''. Since α is unblocked and $\alpha \approx \beta \parallel \gamma$, then $\alpha \approx \beta \parallel \gamma'$ and $\alpha' \approx \beta \parallel \gamma''$, for some l-derivative α' of α; we are assuming that β is strictly unblocked and thus so is α'. Furthermore, since $\alpha \approx \beta \parallel \gamma$ and γ is blocked, there is some l-derivative β' of β such that $\alpha' \approx \beta' \parallel \gamma$ where β' is strictly unblocked because α' is and γ is blocked. Finally, depth$(\alpha') <$ depth(α), and thus by induction we conclude $\gamma \approx \mathbf{0}$, a contradiction.

The proof is complete. □

8.3 ALGEBRAIC CHARACTERISATION

We extend the axiomatic system $\mathcal{A}_{sf}$ with laws regarding the parallel composition operator, and show that the resulting axiomatic system is sound and complete. Notice that we need two expansion laws, since the syntax of finite types does not allow mixing labels and υ in sums, e.g., as in $l.\alpha + \upsilon.\beta$. Moreover, we further need an extra υ-law which allows us to saturate blocked parallel types that do not expand.

PROP/DEFINITION 8.3.1 (AXIOMATISATION).
The laws of Prop/Definition 7.4.1, together with the following laws inductively define the axiomatic system $\mathcal{A}_f$.[2]

CM $\langle \mathcal{T}_f/\approx, \|, \mathbf{0}\rangle$ *is a commutative monoid;*

EXP1 $\sum_{i\in I} \upsilon.\alpha_i \parallel \sum_{j\in J} \upsilon.\beta_j \approx$
$\sum_{i\in I} \upsilon.(\alpha_i \parallel \sum_{j\in J} \upsilon.\beta_j) + \sum_{j\in J} \upsilon.(\sum_{i\in I} \upsilon.\alpha_i \parallel \beta_j)$*;*

EXP2 $\sum_{i\in I} l_i(\widetilde{\alpha}_i).\alpha_i \parallel \sum_{j\in J} m_j(\widetilde{\beta}_j).\beta_j \approx$
$\sum_{i\in I} l_i(\widetilde{\alpha}_i).(\alpha_i \parallel \sum_{j\in J} m_j(\widetilde{\beta}_j).\beta_j) + \sum_{j\in J} m_j(\widetilde{\beta}_j).(\sum_{i\in I} l_i(\widetilde{\alpha}_i).\alpha_i \parallel \beta_j)$*;*

U1 $\upsilon.\sum_{i\in I} \upsilon.\alpha_i \ \approx\ \sum_{i\in I} \upsilon.\alpha_i$*;*

U2 $\upsilon.(\sum_{i\in I} \upsilon.\alpha_i \parallel \sum_{j\in J} m_j(\widetilde{\beta}_j).\beta_j) \approx$
$\upsilon.(\sum_{i\in I} \upsilon.\alpha_i \parallel \sum_{j\in J} m_j(\widetilde{\beta}_j).\beta_j) + \upsilon.(\alpha_k \parallel \sum_{j\in J} m_j(\widetilde{\beta}_j).\beta_j)$*, with* $k \in I$*.*

PROOF. It is straightforward to build the respective bisimulations. □

We ensure first that the rules above are sound.

THEOREM 8.3.1 (SOUNDNESS OF $\mathcal{A}_f$).
If $\vdash \alpha = \beta$ *then* $\alpha \approx \beta$*.*

PROOF. A consequence of Proposition 7.3.4 and of Prop/Definition 8.3.1. □

We now show that the axiomatic system is complete.

[2]Notice that the υ-law of system $\mathcal{A}_{sf}$ is now law U1.

REMARK. To obtain a system that is provably complete, there are three alternatives.

1. Allow arbitrary labelled sums, i.e. mixing labels and υ in sums, and having a single expansion law; the proof of completeness is standard;
2. Add a new inference rule to the equational logic (which should be proved sound), and proceed as usual.

$$\begin{array}{rl} \text{If} & \forall_{i\in I}\exists_{k\in K}\,(\vdash \widetilde{\alpha}_i = \widetilde{\beta}_k,\ l_i \equiv m_k, \text{ and } \vdash \alpha_i \parallel \sum_{j\in J} \upsilon.\alpha_j = \beta_k \parallel \sum_{l\in L} \upsilon.\beta_l) \\ \text{and} & \forall_{k\in K}\exists_{i\in I}\,(\vdash \widetilde{\alpha}_i = \widetilde{\beta}_k,\ l_i \equiv m_k, \text{ and } \vdash \alpha_i \parallel \sum_{j\in J} \upsilon.\alpha_j = \beta_k \parallel \sum_{l\in L} \upsilon.\beta_l) \\ \text{and} & \forall_{j\in J}\exists_{l\in L}\,(\vdash \sum_{i\in I} l_i(\widetilde{\alpha}_i).\alpha_i \parallel \alpha_j = \sum_{k\in K} m_k(\widetilde{\beta}_k).\beta_k \parallel \beta_l) \\ \text{and} & \forall_{l\in L}\exists_{j\in J}\,(\vdash \sum_{i\in I} l_i(\widetilde{\alpha}_i).\alpha_i \parallel \alpha_j = \sum_{k\in K} m_k(\widetilde{\beta}_k).\beta_k \parallel \beta_l), \\ \text{then} & \vdash \sum_{i\in I} l_i(\widetilde{\alpha}_i).\alpha_i \parallel \sum_{j\in J} \upsilon.\alpha_j = \sum_{k\in K} m_k(\widetilde{\beta}_k).\beta_k \parallel \sum_{l\in L} \upsilon.\beta_l\,. \end{array} \tag{8.1}$$

3. Keep the proof system and use induction directly.

The first alternative is somewhat unnatural, since labelled sums represent interfaces of objects, and an arbitrary labelled sum does not represent an interface. Hence, we do not follow this path.

For the last two alternatives, normal forms include a parallel composition, as there is no expansion law for the parallel composition of a labelled sum and a blocked type; thus the proofs of the normal form lemma and of the completeness theorem are different from those for CCS.

The third alternative turns out to be quite difficult (see the Appendix A for a detailed discussion), thus we proceed now according to the second alternative. First we show that the new inference rule is sound.

LEMMA 8.3.2 (SOUNDNESS OF THE NEW INFERENCE RULE).
The inference rule 8.1 is sound.

PROOF. Let $\alpha \equiv \sum_{i\in I} l_i(\widetilde{\alpha}_i).\alpha_i \parallel \sum_{j\in J} \upsilon.\alpha_j$ and $\beta \equiv \sum_{k\in K} m_k(\widetilde{\beta}_k).\beta_k \parallel \sum_{l\in L} \upsilon.\beta_l$. We conduct a case analysis of the possible immediate transitions of α and β.

1. Case $\alpha \xrightarrow{l(\widetilde{\alpha})} \alpha_i \parallel \sum_{j\in J} \upsilon.\alpha_j$.

   Since there is a k such that $l_i \equiv m_k$ and $\widetilde{\alpha}_i \approx \widetilde{\beta}_k$ (as by hypothesis, $\vdash \widetilde{\alpha}_i = \widetilde{\beta}_k$, and the system is sound), then $\beta \xrightarrow{m_k(\widetilde{\beta}_k)} \beta_k \parallel \sum_{l\in L} \upsilon.\beta_l$, and $\alpha_i \parallel \sum_{j\in J} \upsilon.\alpha_j \approx \beta_k \parallel \sum_{l\in L} \upsilon.\beta_l$, as by hypothesis, $\vdash \alpha_i \parallel \sum_{j\in J} \upsilon.\alpha_j = \beta_k \parallel \sum_{l\in L} \upsilon.\beta_l$ and the system is sound.

2. Case $\alpha \xrightarrow{v} \sum_{i\in I} l_i(\widetilde{\alpha}_i).\alpha_i \parallel \alpha_j$.

   There is an l such that $\beta \xrightarrow{v} \sum_{k\in K} m_k(\widetilde{\beta}_k).\beta_k \parallel \beta_l$, and again it follows that $\sum_{i\in I} l_i(\widetilde{\alpha}_i).\alpha_i \parallel \alpha_j \approx \sum_{k\in K} m_k(\widetilde{\beta}_k).\beta_k \parallel \beta_l$.

By symmetry we conclude that $\alpha \approx \beta$. □

Definition 8.3.2 (Normal forms of concurrent finite types).
A type α is a normal form *if it is saturated (cf. Definition 7.4.2), and furthermore, one of the following conditions holds:*

1. $\alpha \equiv \mathbf{0}$;
2. $\alpha \equiv \sum_{i\in I} \upsilon.\alpha_i$ *and each* α_i *is a normal form;*
3. $\alpha \equiv \sum_{i\in I} l_i(\widetilde{\alpha}_i).\alpha_i$ *and each component of each* $\alpha_i\widetilde{\alpha}_i$ *is a normal form;*
4. $\alpha \equiv \alpha_1 \parallel \alpha_2$, *where* α_1 *and* α_2 *are respectively as in 2 and 3 above, with* $\alpha_1, \alpha_2 \not\approx \mathbf{0}$ *and* $I, J \neq \emptyset$.

Lemma 8.3.3 (Normal form lemma).
For all object types α there exists a normal form α' such that $\vdash \alpha = \alpha'$, with $\mathrm{depth}(\alpha') \leq \mathrm{depth}(\alpha)$.

Proof. By induction on the depth of α. The proof is similar to that of Lemma 7.4.2, so we omit the common cases. Hence, we only have two cases to consider.

Case $\alpha \equiv \sum_{i\in I} \upsilon.\alpha_i$.

By induction hypothesis, for each α_i there exists a normal form α'_i such that $\vdash \alpha_i = \alpha'_i$, with $\mathrm{depth}(\alpha'_i) \leq \mathrm{depth}(\alpha_i)$. The interesting case now is when

$$\alpha'_i \equiv (\sum_{j\in J_i} \upsilon.\alpha_j) \parallel \beta, \text{ with } \beta \equiv \sum_{k\in K_i} l_k(\widetilde{\beta}_k).\beta_k \text{ and } J_i, K_i \neq \emptyset\,.$$

Then, we have $\sum_{i\in I} \upsilon.\alpha'_i \xrightarrow{\upsilon.\upsilon} \alpha_j \parallel \beta$, and, by U2, $\vdash \sum_{i\in I} \upsilon.\alpha'_i = \sum_{i\in I} \upsilon.\alpha'_i + \upsilon.(\alpha_j \parallel \beta)$.

Now $(\alpha_j \parallel \beta)$ may not be a normal form (e.g., if $\alpha_j \equiv \mathbf{0}$), but $\mathrm{depth}(\alpha_j \parallel \beta) < \mathrm{depth}(\alpha)$, and thus, by induction hypothesis, there is a normal form γ such that $\vdash \alpha_j \parallel \beta = \gamma$; hence $\vdash \sum_{i\in I} \upsilon.\alpha'_i = \sum_{i\in I} \upsilon.\alpha'_i + \upsilon.\gamma$. We still have to saturate $\upsilon.\gamma$. If it is a blocked sum, using U1 we obtain a saturated form γ'; if it is a parallel composition we use U2 to obtain γ'. So, $\vdash \sum_{i\in I} \upsilon.\alpha'_i = \sum_{i\in I} \upsilon.\alpha'_i + \gamma'$.

Repeatedly applying U2 in the same way to all α'_i of this form, and applying U1 as in the proof of Theorem 7.4.3, we attain a normal form of α, whose depth does not exceed that of $\sum_{i\in I} \upsilon.\alpha'_i$.

Case $\alpha \equiv \gamma_1 \parallel \gamma_2$.

We use the commutative monoid laws and the expansion laws to rewrite α either as a blocked sum, a labelled-prefixed sum, or a parallel composition of the previous two with $I, J \neq \emptyset$, and without increasing the depth of the types. The first two cases are treated as above, and in the parallel composition case we apply the same reasoning to each sum separately, obtaining $\vdash \alpha = \alpha_1 \parallel \alpha_2$, where α_1 and α_2 are respectively a blocked sum and a labelled-prefixed sum, both normal forms. If $\vdash \alpha_1 = \mathbf{0}$ then $\vdash \alpha_1 \parallel \alpha_2 = \alpha_2$; otherwise, $\alpha_1 \parallel \alpha_2$ is a normal form. Moreover, notice that $\text{depth}(\alpha_1) \leq \text{depth}(\sum_{i\in I} \upsilon.\alpha_i)$ and $\text{depth}(\alpha_2) \leq \text{depth}(\sum_{j\in J} l_j(\widetilde{\beta}_j).\beta_j)$. Therefore, $\text{depth}(\alpha_1 \parallel \alpha_2) \leq \text{depth}(\alpha)$.

The proof is complete. □

PROPOSITION 8.3.4.
If $\alpha \equiv \sum_{i\in I} \upsilon.\alpha_i$ is a normal form, then no α_i is a parallel composition.

PROOF. Assume that $\alpha_1 \equiv \sum_{j\in J_i} \upsilon.\alpha_j \parallel \sum_{k\in K_i} l_k(\widetilde{\alpha}_k).\alpha_k$, which is a normal form. Then $\alpha \xrightarrow{\upsilon\upsilon} \alpha_j \parallel \sum_{k\in K_i} l_k(\widetilde{\alpha}_k).\alpha_k$. Since α is saturated it follows that $\alpha \xrightarrow{\upsilon} \alpha_j \parallel \sum_{k\in K_i} l_k(\widetilde{\alpha}_k).\alpha_k$, i.e. for some $i \in I$, $\alpha_i \equiv \alpha_j \parallel \sum_{k\in K_i} l_k(\widetilde{\alpha}_k).\alpha_k$. But then α_j is blocked since α is a normal form, and furthermore, $\alpha_j \not\approx \mathbf{0}$. Since the types are finite, applying repeatedly this procedure lead us to $\alpha_i \equiv \sum_{m\in M_i} l_m(\widetilde{\alpha}_m).\alpha_m \parallel \sum_{k\in K_i} l_k(\widetilde{\alpha}_k).\alpha_k$, for some $i \in I$, which is not a normal form, and we attain an absurd. □

THEOREM 8.3.5 (COMPLETENESS OF $\mathcal{A}_{sf}$).
If $\alpha \approx \beta$ then $\vdash \alpha = \beta$.

PROOF. By induction on the sum of the depths of the types α and β (assumed to be normal forms, by Lemma 8.3.3).

Taking into account the proof of Theorem 7.4.3, we only have one case to consider.

Case $\alpha \equiv \sum_{i\in I} l_i(\widetilde{\alpha}_i).\alpha_i \parallel \sum_{j\in J} \upsilon.\alpha_j$ and $\beta \equiv \sum_{k\in K} m_k(\widetilde{\beta}_k).\beta_k \parallel \sum_{l\in L} \upsilon.\beta_l$.

We examine the indexing sets:

1. If $I \neq \emptyset$, then obviously also $K \neq \emptyset$, as $\alpha \approx \beta$.
2. If $J \neq \emptyset$, then, by Lemma 8.2.3, also $L \neq \emptyset$.
3. Thus, consider $I, J, K, L \neq \emptyset$, and consider the blocked sums not equivalent to $\mathbf{0}$. The proof depends on the transition done by α. There are two cases to consider:

   (a) Case $\alpha \xrightarrow{l_i(\widetilde{\alpha}_i)} \alpha_i \parallel \sum_{j\in J} \upsilon.\alpha_j$.
   Since, by hypothesis, $\alpha \approx \beta$, then there exists a $k \in K$ such that $l_i \equiv m_k$, $\widetilde{\alpha}_i \approx \widetilde{\beta}_k$, and $\beta \xrightarrow{m_k(\widetilde{\beta}_k)} \beta_k \parallel \sum_{l\in L} \upsilon.\beta_l$, with $\alpha_i \parallel \sum_{j\in J} \upsilon.\alpha_j \approx \beta_k \parallel \sum_{l\in L} \upsilon.\beta_l$. By induction hypothesis it follows that $\vdash \widetilde{\alpha}_i = \widetilde{\beta}_k$ and $\vdash \alpha_i \parallel \sum_{j\in J} \upsilon.\alpha_j = \beta_k \parallel \sum_{l\in L} \upsilon.\beta_l$.

(b) case $\alpha \xrightarrow{\upsilon} \sum_{i\in I} l_i(\widetilde{\alpha}_i).\alpha_i \parallel \alpha_j$.

If $\beta \stackrel{\upsilon}{\Longrightarrow} \sum_{k\in K} m_k(\widetilde{\beta}_k).\beta_k \parallel \beta_l$ then, as β is saturated, $\beta \xrightarrow{\upsilon} \sum_{k\in K} m_k(\widetilde{\beta}_k).\beta_k \parallel \beta_l$, and the proof proceeds similarly to the previous case.

Otherwise, $\sum_{i\in I} l_i(\widetilde{\alpha}_i).\alpha_i \parallel \alpha_j \approx \sum_{k\in K} m_k(\widetilde{\beta}_k).\beta_k \parallel \sum_{l\in L} \upsilon.\beta_l$ with the sum of their depths being lesser than the sum of the original depths, and we can again use the induction hypothesis.

The result for the case we are examining follows by the inference rule 8.1.

□

The appendix presents a possible alternative proof of this result that does not require the new inference rule.

Chapter 9

Behavioural types

Finally, we present the *Algebra of Behavioural Types*, ABT for short. We obtain it by extending the algebra of concurrent finite types with a recursive operator μ to denote (regular) infinite types that enable us to characterise the behaviour of persistent objects, as well as that of objects living in methods of persistent objects.

The axiomatisation of the notion of type equivalence requires three more new laws:

1. the first says that unfolding recursive types preserves lsb;
2. the second says that equalities involving recursive types have unique solutions up to lsb;
3. and the third allows the saturation of recursive types.

We conclude that the absence of mixed sums in ABT leads to a simpler axiomatic system than that of CCS.

In this chapter, we present the syntax and operational semantics of ABT, add to the axiomatic system of the previous chapter three new laws regarding the behaviour of recursive types, laws that we prove correct. The next chapter shows that the axiomatic system is complete for image-finite types.

9.1 Syntax

Assume a countable set of variables, denoted by t and possibly subscripted or primed, disjoint from the set of method names considered in the previous chapters.

Definition 9.1.1 (Behavioural types).
The grammar below defines the set $\mathcal{T}$ of behavioural types.

$$\alpha ::= \sum_{i \in I} l_i(\widetilde{\alpha}_i).\alpha_i \mid \sum_{i \in I} \upsilon.\alpha_i \mid \alpha \parallel \alpha \mid t \mid \mu t.\alpha$$

where I *is a finite, possibly empty, indexing set, and $\widetilde{\alpha}$ is a finite sequence of types.*

Since types may have type variables, we define when does a type variable occur free in a type, when does it occur bound.

Definition 9.1.2 (Free and bound variables). *An occurrence of the variable t in the part α of the type $\mu t.\alpha$ is* bound*; otherwise the occurrence of t is* free.

Alpha-conversion in a type $\mu t.\alpha$ is defined as usual. We henceforth assume that types do not have free type variables.

Notation. To simplify our work, we use the following conventions.

1. Assume a variable convention like in Barendregt [Bar84], and that types are equal up-to alpha-conversion.
2. The type $\alpha[\beta/t]$ denotes the substitution in α of β for the free occurrences of t.

   Moreover, $\mathrm{fv}(\alpha)$ denotes the set of variables that occur free in α and $\mathrm{v}(\alpha)$ denotes the set of all variables of the type α.
3. Let $\{\tilde{t}\}$ denote the set of the elements of the sequence $\tilde{t}$.
4. For simplicity, we sometimes write $t = \alpha$ instead of $\mu t.\alpha$, and $\alpha(\beta)$ instead of $\alpha[\beta/t]$. This makes the presentation of some proofs considerably lighter.

9.2 Operational semantics

Assume the set of labels defined in Definition 7.2.2. A labelled transition relation defines the operational semantics of the algebra. It is obtained from that of the previous section by adding a new rule. Table 9.1 presents all the rules together.

$$\text{ACT} \quad \textstyle\sum_{i\in I} \pi_i.\alpha_i \xrightarrow{\pi_j} \alpha_j \quad (j \in I)$$

$$\text{RPAR} \ \frac{\alpha \xrightarrow{\pi} \alpha'}{\alpha \parallel \beta \xrightarrow{\pi} \alpha' \parallel \beta} \qquad \text{LPAR} \ \frac{\alpha \xrightarrow{\pi} \alpha'}{\beta \parallel \alpha \xrightarrow{\pi} \beta \parallel \alpha'} \qquad \text{REC} \ \frac{\alpha[\mu t.\alpha/t] \xrightarrow{\pi} \alpha'}{\mu t.\alpha \xrightarrow{\pi} \alpha'}$$

Table 9.1: The labelled transition relation of the Algebra of Behavioural Types.

DEFINITION 9.2.1 (LABELLED TRANSITION RELATION).
The axiom schema of Definition 7.2.2 together with the two rules of Definition 8.2.1 and with the rule below inductively define the labelled transition relation *of the algebra of behavioural types.*

$$\text{REC} \ \frac{\alpha[\mu t.\alpha/t] \xrightarrow{\pi} \alpha'}{\mu t.\alpha \xrightarrow{\pi} \alpha'}$$

Notice that $\mu t.\alpha$ and $\alpha[\mu t.\alpha/t]$ have the same transitions, and thus are bisimilar.

LEMMA 9.2.1.
$\mu t.\alpha \approx \alpha[\mu t.\alpha/t]$.

PROOF. Immediate. □

It is simple to verify, using standard techniques, that *lsb* is still a congruence.

PROPOSITION 9.2.2 (CONGRUENCE).
The recursive operator preserves label-strong bisimulation.

PROOF. We show that the relation $\{(\mu t.\alpha, \mu t.\beta) \mid \alpha \approx \beta\}$ is a bisimulation, and thus, μt preserves $\approx$. The proof is by induction on the maximum length of the derivation of the transition (i.e., by 'transition induction' — see [Mil89a]). The base case is trivial. There are two cases to be considered in the induction step.

1. Let $\mu t.\alpha \xrightarrow{l(\widetilde{\alpha})} \alpha'$.

   Depending on the form of α, there are several cases to consider.

   Case $\alpha \equiv \sum_{i\in I} l_i(\widetilde{\alpha}_i).\alpha_i$.

   Since $\mu t.\alpha \approx \alpha[\mu t.\alpha/t]$, $\alpha[\mu t.\alpha/t] \equiv \sum_{i\in I} l_i(\widetilde{\alpha}[\mu t.\alpha/t]).\alpha_i[\mu t.\alpha/t]$, with $\widetilde{\alpha} \approx \widetilde{\alpha}[\mu t.\alpha/t]$ and $\alpha' \approx \alpha_i[\mu t.\alpha/t]$, for some $i \in I$. As $\alpha \approx \beta$ by hypothesis, also $\beta \equiv \sum_{j\in J} l_j(\widetilde{\beta}).\beta_j$ and there is a $j \in J$ such that $\alpha_i \approx \beta_j$ and $\widetilde{\alpha} \approx \widetilde{\beta}$, thus $\widetilde{\alpha}[\mu t.\alpha/t] \approx \widetilde{\beta}[\mu t.\beta/t]$. Hence, $\mu t.\beta \xrightarrow{l(\widetilde{\beta})} \beta'$, and since by induction hypothesis $\alpha_i[\mu t.\alpha/t] \approx \beta_j[\mu t.\beta/t]$, the result follows by taking $\alpha' \equiv \alpha_i[\mu t.\alpha/t]$ and $\beta' \equiv \beta_j[\mu t.\beta/t]$.

The remaining cases are straightforward; one simply has to use the induction hypothesis.

2. Let $\mu t.\alpha \xrightarrow{v} \alpha'$.

   The proof is as in the first case, except that we do not need to worry about the parameters in the prefixes.

The proof is complete. □

9.3 Subtyping

The *principle of substitutability* states that "a type β is a subtype of a type α, if β can safely substitute α". "Safely" means that no run-time error arises. Therefore, subtyping allows the substitution of: (1) a type for one with less methods (co-variant in width), as it is safe to provide more than it is expected; and (2) a parameter type for one with more methods (contra-variant in the arguments), as it is safe to assume that that the argument has less behaviour than it really has.

Instead of defining the subtyping relation via typing rules we propose a semantic definition. Notice that our equivalence notion *lsb* is a symmetric *simulation*.

Definition 9.3.1 (Similarity on types).

1. *A binary relation $R \subseteq \mathcal{T} \times \mathcal{T}$ is a* label-strong simulation*, or simply a simulation, if whenever $\alpha\, R\, \beta$ we have:*
   (a) $\beta \xrightarrow{l(\widetilde{\beta})} \beta'$ implies $\exists \alpha', \widetilde{\alpha}$ ($\alpha \xrightarrow{l(\widetilde{\alpha})} \alpha'$ and $\alpha'\widetilde{\beta}\, R\, \beta'\widetilde{\alpha}$);
   (b) $\beta \xrightarrow{v} \beta'$ implies $\exists \beta'$ ($\alpha \Longrightarrow \alpha'$ and $\alpha'\, R\, \beta'$);
2. *Type β is* label-strong similar *to type α, or α* simulates *β, and we write $\alpha \leq \beta$, if there is a label-strong simulation R such that $\alpha\, R\, \beta$.*

A symmetric simulation is a label-strong bisimulation (Definition 7.3.1 in page 50). The simulation can be seen as a *subtyping* relation, since it is a quasi-order (reflexive and transitive, but not necessarily anti-symmetric). Thus, if α *simulates* α', we say that α is a *subtype* of α', and write $\alpha \leq \alpha'$.

Example 9.3.2.

1. $(n \parallel l(m)) \leq l(m)$ *and* $(n + l(m)) \leq l(m)$;
2. $l(m) \leq l(m + n)$;
3. $l(m) \leq v.l(m)$.

Proposition 9.3.1 (($\mathcal{T}, \leq$) is a quasi-ordered set).

1. *$\alpha \leq \alpha$;*
2. *if $\alpha \leq \beta$ and $\beta \leq \gamma$ then $\alpha \leq \gamma$;*

Proof. Straightforward, simply using the definition □

The operators of ABT, as well as *lsb*, preserve the simulation relation.

Proposition 9.3.2 (Congruence).

1. *Similarity is a congruence relation;*
2. *lsb preserves similarity.*

Proof. The proof of the first clause is standard. The second clause is trivial, since $\approx$ implies $\leq$. □

Adding anti-symmetry to the quasi-order ($\mathcal{T}, \leq$), one obtains an pre-order relation that induces an equivalence relation (if $\alpha \leq \beta$ and $\beta \leq \alpha$ then $\alpha = \beta$). This relation is coarser than *lsb*, since usually there are types that can simulate each other without being bisimilar. A simple example is the pair of types $\upsilon.(a+b)$ and $\upsilon.(\upsilon.a + \upsilon.(a+b))$.

9.4 Axiomatic system

We present an axiomatisation of the equivalence notion, adding two recursion rules to the previous axiomatic system, and show its soundness.

Definition 9.4.1 (Guarded variables).
A free variable t is guarded in α *if all its occurrences are within some label-prefixed subexpression of α. We say α* is guarded *if all its free variables are guarded.*

Example 9.4.2. The variable t is guarded in $l(t).\alpha$, in $l(\widetilde{\alpha}).t$, and in $l(\upsilon.t).\mathbf{0}$, but not in $\upsilon.t$.

Prop/Definition 9.4.3 (The axiomatic system $\mathcal{A}$).
The laws of Prop/Definition 8.3.1, together with the following recursion laws, inductively define the axiomatic system $\mathcal{A}$.

R1 $\mu t.\alpha \approx \alpha[\mu t.\alpha/t]$;

R2 *if $\beta \approx \alpha[\beta/t]$ then $\beta \approx \mu t.\alpha$, provided that α is guarded;*

R3 $\mu t.(\upsilon.t + \sum_{i\in I} \upsilon.\alpha_i) \approx \mu t.\sum_{i\in I} \upsilon.\alpha_i$.

The soundness of the axioms above is not a trivial result. To prove it, one has first to ensure that the equations have unique solutions, i.e., if β is guarded, $\alpha_1 \approx \beta[\alpha_1/t]$, and $\alpha_2 \approx \beta[\alpha_2/t]$ then $\alpha_1 \approx \alpha_2$. The next section is dedicated to the proof of that result. Once we establish that, the proof of the soundness of the axioms follows.

Proof. Law R2 of Prop/Definition 9.4.3 is a corollary of this result and of law R1 (which has a straightforward proof). To prove law R3, one simply has to build the respective bisimulation. □

Remark. Laws R3 and R5 of CCS have no correspondence in this setting, as ABT does not have binary sums. Thus, our law R3 corresponds to (is an instance of) the CCS's law R4.

9.5 Unique solutions

The proof follows a method analogous to that used for CCS. We need the auxiliary notion of *lsb* up to $\approx$.

Definition 9.5.1 (Lsb up to $\approx$).
An lsb up to $\approx$ *is a symmetric binary relation R on types such that, whenever $\alpha\ R\ \beta$ then*

1. $\alpha \xrightarrow{l(\widetilde{\alpha})} \alpha'$ *implies* $\exists_{\widetilde{\beta},\beta'}\ \beta \xrightarrow{l(\widetilde{\beta})} \beta'$ *and* $\alpha'\widetilde{\alpha} \approx R \approx \beta'\widetilde{\beta}$, *and*
2. $\alpha \Longrightarrow \alpha'$ *implies* $\exists_{\beta'}\ \beta \Longrightarrow \beta'$ *and* $\alpha' \approx R \approx \beta'$ *.*

We show that *lsb* up to $\approx$ is still a label-strong bisimulation.

Proposition 9.5.1 (Lsb up to $\approx$ is a bisimulation).
Let R be an lsb up to $\approx$. Then,

1. $\approx R \approx$ *is an lsb.*
2. $R \subseteq \approx$*.*

Proof. Similar to the proof for weak bisimulation up to weak bisimilarity in [Mil89a]. □

Remark. With $\xrightarrow{v}$ instead of $\Longrightarrow$ in the antecedent of the second condition of Definition 9.5.1, the previous proposition would not hold, as the example $R = \{(v.v.a.\mathbf{0}, v.\mathbf{0})\}$ shows (notice the similarities with the Exercise 5.14 in [Mil89a]).

Two technical lemmas are necessary to prove the result. We give them below, and the theorem follows.

LEMMA 9.5.2.
Let α be guarded with free variables in $\{\tilde{t}\}$.

1. *If $\alpha(\widetilde{\beta}) \xrightarrow{l(\widetilde{\gamma})} \gamma$ then there exist α' and $\widetilde{\alpha}$ with free variables in $\tilde{t}$ such that $\gamma \equiv \alpha'(\widetilde{\beta})$ and $\widetilde{\gamma} \equiv \widetilde{\alpha}(\widetilde{\beta})$, and, for all $\widetilde{\beta}'$, $\alpha(\widetilde{\beta}') \xrightarrow{l(\widetilde{\alpha}(\widetilde{\beta}'))} \alpha'(\widetilde{\beta}')$.*
2. *If $\alpha(\widetilde{\beta}) \xrightarrow{v} \gamma$ then there exists α' with free variables in $\tilde{t}$ such that $\gamma \equiv \alpha'(\widetilde{\beta})$ and, for all $\widetilde{\beta}'$, we have $\alpha(\widetilde{\beta}') \xrightarrow{v} \alpha'(\widetilde{\beta}')$. Furthermore, α' is guarded.*

PROOF. By induction on the maximum length of the derivation of the transition. There are two cases to consider, depending on the transition performed.

1. Let $\alpha(\widetilde{\beta}) \xrightarrow{l(\widetilde{\gamma})} \gamma$. The base case is trivial. For the induction step, we have three different cases to consider, according to the possible forms of α.
   (a) Case $\alpha \equiv \sum_{i \in I} l_i(\widetilde{\alpha}_i).\alpha_i$.
   Then $\alpha(\widetilde{\beta})$ is $\sum_{i \in I} l_i(\widetilde{\alpha}_i(\widetilde{\beta})).\alpha_i(\widetilde{\beta})$, and thus $l(\widetilde{\gamma})$ is $l_i(\widetilde{\alpha}_i(\widetilde{\beta}))$ and γ is $\alpha_i(\widetilde{\beta})$ for some $i \in I$. The result follows from taking $\alpha' \equiv \alpha_i$ and $\widetilde{\alpha} \equiv \widetilde{\alpha}_i$.
   (b) Case $\alpha \equiv \alpha_1 \parallel \alpha_2$.
   Then $\alpha(\widetilde{\beta}) \equiv \alpha_1(\widetilde{\beta}) \parallel \alpha_2(\widetilde{\beta})$. There are two sub-cases to consider:
   either $\gamma \equiv \gamma_1 \parallel \alpha_2(\widetilde{\beta})$, with $\alpha_1(\widetilde{\beta}) \xrightarrow{l(\widetilde{\gamma})} \gamma_1$,
   or $\gamma \equiv \alpha_1(\widetilde{\beta}) \parallel \gamma_2$, with $\alpha_2(\widetilde{\beta}) \xrightarrow{l(\widetilde{\gamma})} \gamma_2$, by a shorter derivation.
   Without loss of generality, assume the first case. Since α is guarded, so is α_1, and thus, by induction hypothesis, it follows that $\gamma_1 \equiv \alpha_1'(\widetilde{\beta})$ and $\widetilde{\gamma} \equiv \widetilde{\alpha}_1(\widetilde{\beta})$. The result follows from taking $\alpha' \equiv \alpha_1' \| \alpha_2$ and $\widetilde{\alpha} \equiv \widetilde{\alpha}_1$.
   (c) Case $\alpha \equiv \mu t.\beta$.
   Then $\alpha(\widetilde{\beta}) \equiv \mu t.\beta(\widetilde{\beta})$, where the free variables of β are taken from t and $\tilde{t}$. The variables $\tilde{t}$ must be guarded in β, otherwise they would not be guarded in α. If $\mu t.\beta(\widetilde{\beta}) \xrightarrow{l(\widetilde{\gamma})} \gamma$ then $\beta[\mu t.\beta(\widetilde{\beta})/t] \xrightarrow{l(\widetilde{\gamma})} \gamma$, by a shorter derivation. But the variables of $\tilde{t}$ are guarded in $\beta[\mu t.\beta/t]$, and thus, by induction hypothesis, we conclude that $\gamma \equiv \alpha'(\widetilde{\beta})$ and $\widetilde{\gamma} \equiv \widetilde{\alpha}(\widetilde{\beta})$.
2. Let $\alpha(\widetilde{\beta}) \xrightarrow{v} \gamma$. The proof is as in the first case, except that we need not worry about parameters in prefixes. The main difference is that we must also prove that α' is guarded. The base case is trivial. Case $\alpha \equiv \sum_{i \in I} v.\alpha_i$, all the α_i must be guarded because α is, and thus α' is guarded. In the remaining cases, the conclusion is a consequence of the induction hypothesis.

$\square$

LEMMA 9.5.3.
Let α be guarded with free variables in $\tilde{t}$. If $\alpha(\widetilde{\beta}) \Longrightarrow \gamma$ then there exists α' with free variables in $\tilde{t}$ such that $\gamma \equiv \alpha'(\widetilde{\beta})$ and, for all $\widetilde{\beta}'$, we have $\alpha(\widetilde{\beta}') \Longrightarrow \alpha'(\widetilde{\beta}')$.

PROOF. Let $\alpha(\widetilde{\beta}) \Longrightarrow \gamma$ and let n be the actual number of υ's in the transition. The proof follows easily from the previous lemma, by induction on n. $\square$

We are finally in a position to prove the main result for infinite types: the theorem of the uniqueness of the solutions of equations.

THEOREM 9.5.4 (UNIQUE SOLUTIONS OF EQUATIONS).
Let $\widetilde{\beta}$ be guarded, $\widetilde{\alpha}_1 \approx \widetilde{\beta}(\widetilde{\alpha}_1)$ and $\widetilde{\alpha}_2 \approx \widetilde{\beta}(\widetilde{\alpha}_2)$. Then $\widetilde{\alpha}_1 \approx \widetilde{\alpha}_2$.

PROOF. Let R be the relation $\{(\gamma(\widetilde{\alpha}_1), \gamma(\widetilde{\alpha}_2)) \mid \mathrm{v}(\gamma) \subseteq \tilde{t}\}$. We will show that:

1. $\gamma(\widetilde{\alpha}_1) \xrightarrow{l(\widetilde{\delta}_1)} \alpha_1$ implies $\exists \alpha_2, \widetilde{\delta}_2\, \gamma(\widetilde{\alpha}_2) \xrightarrow{l(\widetilde{\delta}_2)} \alpha_2$ and $\alpha_1(\widetilde{\delta}_1) \approx R \approx \alpha_2(\widetilde{\delta}_2)$;
2. $\gamma(\widetilde{\alpha}_1) \Longrightarrow \alpha_1$ implies $\exists \alpha_2\, \gamma(\widetilde{\alpha}_2) \Longrightarrow \alpha_2$ and $\alpha_1 \approx R \approx \alpha_2$.

By the results we have seen before, and by symmetry, this establishes that R is a label-strong bisimulation up to label-strong bisimilarity, and also that $\gamma(\widetilde{\alpha}_1) \approx \gamma(\widetilde{\alpha}_2)$ for all γ, which includes the cases $\alpha_{1i} \approx \alpha_{2i}$ $(\gamma \equiv t_i)$.
So let us prove 1. Since $\approx$ is a congruence, $\gamma(\widetilde{\alpha}_1) \approx \gamma(\widetilde{\beta}(\widetilde{\alpha}_1))\; R\; \gamma(\widetilde{\beta}(\widetilde{\alpha}_2)) \approx \gamma(\widetilde{\alpha}_2)$, and, by hypothesis, $\widetilde{\alpha}_1 \approx \widetilde{\beta}(\widetilde{\alpha}_1)$ and $\widetilde{\beta}(\widetilde{\alpha}_2) \approx \widetilde{\alpha}_2$. If $\gamma(\widetilde{\alpha}_1) \xrightarrow{l(\widetilde{\delta}_1)} \alpha_1$ then $\gamma(\widetilde{\beta}(\widetilde{\alpha}_1)) \xrightarrow{l(\widetilde{\delta}'_1)} \alpha'_1$, with $\widetilde{\delta}_1 \approx \widetilde{\delta}'_1$ and $\alpha_1 \approx \alpha'_1$. By Lemma 9.5.2, there are $\widetilde{\gamma}$ and γ' such that $\widetilde{\delta}'_1 \equiv \widetilde{\gamma}(\widetilde{\alpha}_1)$, $\alpha'_1 \equiv \gamma'(\alpha_1)$ and $\gamma(\widetilde{\beta}(\widetilde{\alpha}_2)) \xrightarrow{l(\widetilde{\gamma}(\widetilde{\alpha}_2))} \alpha'_2 \equiv \gamma'(\widetilde{\alpha}_2)$, which implies $\widetilde{\gamma}(\widetilde{\alpha}_2) \xrightarrow{l(\widetilde{\delta}_2)} \alpha_2$, with $\widetilde{\gamma}(\widetilde{\alpha}_2) \approx \widetilde{\delta}_2$ and $\alpha'_2 \approx \alpha_2$. Hence, $\widetilde{\delta}_1 \approx R \approx \widetilde{\delta}_2$ and $\alpha_1 \approx R \approx \alpha_2$.

We prove 2 by similar reasoning, but using Lemma 9.5.3, instead of Lemma 9.5.2. $\square$

Chapter 10

Completeness for image-finite types

The presence of the recursive operator in the algebra allows us to define image-finite types like $\mu t.(l.t)$, a type that represents an infinite sequence of l-actions. It represents the behaviour of a persistent object that repeatedly offers a method l.

The recursive operator, together with the parallel composition operator, allows us to define image-infinite types like $\mu t.v.(l \| t)$. This is the type of an ephemeral object with a method l that lives inside a method of a persistent object.

It is well known that a process algebra where it is possible to define such terms does not have complete axiomatisations of equivalence notions, at least if it has full computational power. Therefore, we show in this section that the axiom system of the previous section is complete for image-finite types (dynamic types without the parallel operator), and discuss in the conclusions the possibility of a completeness result for the full type algebra.

The completeness proof of the axiomatisation of the equivalence notion for image-finite dynamic types is not trivial, and so we dedicate to it the next chapter. The proof follows the structure of that for image-finite CCS [Mil89b], being significantly simpler.

10.1 TRANSFORMING SETS OF EQUATIONS

The first step is the transformation of recursive types into sets of equations, which are easier to manipulate. Henceforth, consider that types are not constructed with the recursive operator μ.

TERMINOLOGY. Consider a set of variables $\{\tilde{t}\}$, and a set of types $\{\widetilde{\beta}\}$ where $\mathrm{fv}(\beta) \subseteq \tilde{t}$. Let $S : \tilde{t} = \widetilde{\beta}$ be a system of formal equations, with $\mathrm{v}(S)$ denoting its set of variables.

1. We write $t_i \xrightarrow{\pi} t$, if $\pi.t$ is a summand of β_i.
2. System S is guarded, if $\forall_i\ t_i \stackrel{v}{\not\Longrightarrow} t_i$ and it is *saturated* if $t \stackrel{v}{\Longrightarrow} t'$ implies $t \xrightarrow{v} t'$.
3. System S is standard, if $\beta_i \equiv \sum_{j\in J} \upsilon.t_{f(i,j)}$ or $\beta_i \equiv \sum_{j\in J} l_{ij}(\tilde{t}'_{(i,j)}).t_{f(i,j)}$, where $\tilde{t}'_{(i,j)} \subseteq \tilde{t}$.
4. Let $\mathrm{fv}(\alpha_1) = \emptyset$. We write $\alpha_1 \Vdash S$ if there is $\widetilde{\alpha} = \alpha_1 \cdots \alpha_n$, such that $\vdash \widetilde{\alpha} = \widetilde{\beta}[\widetilde{\alpha}/\tilde{t}]$.

The following lemma is crucial to prove the theorem that ensures that equations have unique solutions up to *lsb*.

LEMMA 10.1.1 (SATURATION).
Let $\alpha \Vdash S$, with S standard and guarded. There is an S' standard, guarded, and saturated such that $\alpha \Vdash S'$.

PROOF. Similar to the proof of Lemma 3.1 in [Mil89b]. □

THEOREM 10.1.2.
Consider $S : \tilde{t} = \widetilde{\gamma}$ and $T : \tilde{u} = \widetilde{\delta}$ where $\tilde{t} = \{t_1, \ldots, t_n\}$ with $\tilde{t} = \mathrm{v}(S)$, and $\tilde{u} = \{u_1, \ldots, u_n\}$ with $\tilde{u} = \mathrm{v}(T)$, and let $\tilde{t}$ and $\tilde{u}$ be disjoint. Let $\alpha \Vdash S$, and $\beta \Vdash T$, with S and T standard and guarded, and let $\alpha \approx \beta$. Then there is U standard and guarded such that $\alpha \Vdash U$ and $\beta \Vdash U$.

PROOF. By lemma 10.1.1 we assume S and T saturated.
There are $\widetilde{\alpha}$ and $\widetilde{\beta}$, with $\alpha_1 \equiv \alpha$ and $\beta_1 \equiv \beta$ such that $\vdash \widetilde{\alpha} = \widetilde{\gamma}[\widetilde{\alpha}/\tilde{t}]$ and $\vdash \widetilde{\beta} = \widetilde{\delta}[\widetilde{\beta}/\tilde{u}]$.
Since $\alpha \approx \beta$ then:

1. $t_1 \xrightarrow{l(\tilde{t}')} t_i$ implies $\exists_{u_j,\tilde{u}'}(u_1 \xrightarrow{l(\tilde{u}')} u_j$ and $\alpha_i\widetilde{\alpha}' \approx \beta_j\widetilde{\beta}')$[1];
2. $u_1 \xrightarrow{l(\tilde{u}')} u_j$ implies $\exists_{t_i,\tilde{t}'}(t_1 \xrightarrow{l(\tilde{t}')} t_i$ and $\alpha_i\widetilde{\alpha}' \approx \beta_j\widetilde{\beta}')$;
3. $t_1 \xrightarrow{v} t_i$ implies $\exists_{u_j}(u_1 \xrightarrow{v} u_j$ and $\alpha_i \approx \beta_j)$;
4. $u_1 \xrightarrow{v} u_j$ implies $\exists_{t_i}(t_1 \xrightarrow{v} t_i$ and $\alpha_i \approx \beta_j)$.

[1]Consider $\widetilde{\alpha}'$ a subfamily of types corresponding to $\tilde{t}'$.

Consider the following bisimulation relation R:

1. $R \subseteq \tilde{t} \times \tilde{u}$ such that
   (a) $t \xrightarrow{l(\tilde{t}')} t'$ implies $\exists_{u',\tilde{u}'}(u \xrightarrow{l(\tilde{u}')} u'$ and $t'\tilde{t}' \, R \, u'\tilde{u}')$;
   (b) $u \xrightarrow{l(\tilde{u}')} u'$ implies $\exists_{t',\tilde{t}'}(t \xrightarrow{l(\tilde{t}')} t'$ and $t'\tilde{t}' \, R \, u'\tilde{u}')$;
   (c) $t \xrightarrow{\upsilon} t'$ implies ($t' \, R \, u$ or $\exists_{u'}(u \xrightarrow{\upsilon} u'$ and $t' \, R \, u'$));
   (d) $u \xrightarrow{\upsilon} u'$ implies ($t \, R \, u'$ or $\exists_{t'}(t \xrightarrow{\upsilon} t'$ and $t' \, R \, u'$)).
2. $t_1 \, R \, u_1$.

We aim at $U : \tilde{v} = \tilde{\varepsilon}$, where $\tilde{v} = \{v_{ij} \mid t_i \; R \; u_j\}$, and $\tilde{\varepsilon} = \{\varepsilon_{ij} \mid t_i \; R \; u_j\}$, with ε_{ij} being a sum with summands:

1. $l(\tilde{v}').v_{kl}$, if $t_i \xrightarrow{l(\tilde{t}')} t_k$ and $u_j \xrightarrow{l(\tilde{u}')} u_l$ and $t_k\tilde{t}' \, R \, u_l\tilde{u}'$;[2]
2. $\upsilon.v_{kl}$, if $t_i \xrightarrow{\upsilon} t_k$ and $u_j \xrightarrow{\upsilon} u_l$ and $t_k \, R \, u_l$.

To prove '$\alpha \Vdash U$', with v_{11} being the leading variable, we must find $\widetilde{\varphi}$ such that $\varphi_1 \equiv \alpha$ and $\vdash \widetilde{\varphi} = \tilde{\varepsilon}[\widetilde{\varphi}/\tilde{v}]$. Let $\varphi_{ij} \equiv \alpha_i$. Since $t_i \approx u_j$ we have two cases to consider:

1. Case $t_i \xrightarrow{l(\tilde{t}')} t_k$ and $t_k \, R \, u_l$ and $\tilde{t}' \, R \, \tilde{u}$; then ε_{ij} is a labelled sum with a summand $l(\tilde{v}').v_{kl}$, and α_i has a summand $l(\alpha', \dots).\alpha_k$, thus $\varepsilon_{ij}[\widetilde{\varphi}/\tilde{v}]$ has a summand $l(\alpha', \dots).\alpha_k$, and we conclude the equality, since ε_{ij} is a labelled sum.
2. Case $t_i \xrightarrow{\upsilon} t_k$ and $t_k \, R \, u_l$; then ε_{ij} is a blocked sum with a summand $\upsilon.v_{kl}$, and α_i has a summand $\upsilon.\alpha_k$, thus $\varepsilon_{ij}[\widetilde{\varphi}/\tilde{v}]$ has a summand $\upsilon.\alpha$, with $\upsilon.t_k \approx \upsilon.u_l$, and we conclude the equality, since ε_{ij} is a blocked sum.

The proof is complete. □

10.2 Completeness for image-finite guarded types

From the results in the previous section we establish the main result of this chapter: the completeness of the axiom system $\mathcal{A}$ with respect to the equivalence notion for image-finite guarded types, that is, types without the parallel composition operator.

PROPOSITION 10.2.1.
For every image-finite guarded type α with free variables $\tilde{t}$ there is S standard and guarded such that $\alpha \Vdash S$, and $\mathrm{fv}(S) \subseteq \tilde{t}$. Moreover, if t is guarded in α, then t is guarded in S.

[2]Consider v'_n the variable associated with $t'_n \times u'_n$, e.g., $v'_n = v_{25}$ if $t'_n = t_2$ and $u'_n = u_5$.

Proof. Similar to the proof of Theorem 4.1 in [Mil89b]. □

Theorem 10.2.2 (Unique solution of equations).
If S is guarded with free variables $\tilde{t}$, then there is a type α such that $\alpha \Vdash S$. Moreover, if for some β with free variables $\tilde{t}$, $\beta \Vdash S$, then $\vdash \alpha = \beta$.

Proof. Similar to the proof of Theorem 4.2 in [Mil89b]. □

Theorem 10.2.3 (Completeness).
If α and β are image-finite guarded types, and $\alpha \approx \beta$ then $\vdash \alpha = \beta$.

Proof. Similar to the proof of Theorem 4.3 in [Mil89b]. □

10.3 Completeness for image-infinite types

Completeness for infinite state types is a considerably more difficult problem. One cannot hope for completeness of axiomatisations of equivalence notions in CCS for the calculus is computationally complete. Since the full computational power comes from the substitution mechanisms (communication in this case), in calculi without communication it still makes sense to look for completeness. The study of image-infinite (or infinite-state) systems is a lively area of concurrency theory, with several important results established [BE97, CH93, Mol96]. We focus our attention in two process algebras: BPA and BPP. BPA is the class of Basic Process Algebra of Bergstra and Klop [BK85], corresponding to the transition systems associated with Greibach Normal Form (GNF) context-free grammars, in which only left-most derivations are allowed. BPP is the class of Basic Parallel Processes of Christensen [Chr93], which is the parallel counterpart of BPA but with arbitrary derivations. Strong bisimilarity is decidable for BPA [CHS95] and for BPP [CHM93b, CHM93a]. However there is still no such result for weak bisimilarity on full BPA and BPP, although the result is already established for the totally normed subclasses [Hir97], and a possible decision procedure for full BPP is NP-hard [Stř98].

Nevertheless, we are looking for completeness, since decidability is stronger than what we need: the existence of a proof for each equation suffices.

CHAPTER 11

FINAL DISCUSSION

To represent the behaviour of a concurrent object with several co-existing clones (the model emerging naturally from the calculus in its full generality) via a type, a process algebra is a natural idea.

Since a type (partially) specifies an object, there are three basic requirements:

1. method calls are the basic actions, and thus,
2. the silent action is external rather than internal, as it corresponds to an action in another object (not directly observable);
3. sums (records-as-objects) are either prefixed by method calls or by the silent action, i.e. there are no free sums.

Hence, actions are either method calls or the silent action, the basic process is the (possibly empty) action-prefixed sum, and the composition operators are parallel composition and recursion.

No existing process algebra has all these characteristics together. BPP is similar to, but not exactly, what we need. For the sake of simplicity and to avoid confusion we define a new process algebra, ABT.

The notion of observation differs from other works on process algebra. It leads to a new and simple notion of equivalence, *lsb*. To conclude this part about the type algebra we discuss two last questions:

1. why is *lsb* our notion of type equivalence?
2. what else remains to be done?

The remaining sections of this chapter are about these topics.

11.1 The notion of bisimulation

Why is *lsb* our equivalence notion? Could it be different? Could we have used an existent notion? We now approach these questions.

11.1.1 An alternative

An alternative notion of bisimulation is below.

DEFINITION 11.1.1.

1. *A symmetric binary relation* $R \subseteq \mathcal{T} \times \mathcal{T}$ *is a* label-semi-strong bisimulation*, (lssb), if whenever* $\alpha \, R \, \beta$ *then*

   (a) $\alpha \xrightarrow{l(\widetilde{\alpha})} \alpha'$ *implies* $\exists \beta', \widetilde{\beta}, \gamma \; (\beta \xrightarrow{l(\widetilde{\beta})} \gamma \Longrightarrow \beta'$ *and* $\alpha' \widetilde{\alpha} \, R \, \beta' \widetilde{\beta})$*;*
   (b) $\alpha \xrightarrow{v} \alpha'$ *implies* $\exists \beta' \; (\beta \Longrightarrow \beta'$ *and* $\alpha' \, R \, \beta')$*;*

2. *Two types* α *and* β *are* label-semi-strong bisimilar*, and we write* $\alpha \approx_s \beta$*, if there is a label-semi-strong bisimulation* R *such that* $\alpha \, R \, \beta$*.*

Again, $\approx_s$ is an equivalence relation and $\alpha \approx_s \beta$ holds if and only if conditions 1(a) and 1(b) of the previous definition hold with R replaced by $\approx_s$. Furthermore, *lssb* is a congruence relation.

This notion differs from *lsb* by allowing unblockings after method calls (condition 1(a)). For deterministic finite types the two notions coincide, as we have previously shown [RRV98]. However, as we discuss in that paper, the notions do not coincide in more general transition systems, namely in non-deterministic ones.

Take the systems in Figure 11.1. In $l.v.l$, the second l is only observable after the occurrence of the unblocking, which corresponds to the execution of some action in another object. There is a causal dependency between the first l, the action corresponding to the unblocking, and the second l. If the law $l.v.l = l.v.l + l.l$ holds for some equivalence notion then the notion does not capture causality between action execution in different objects, and thus it is a *local* notion, whereas a notion that distinguishes the types in the law is *global* (with respect to the community of objects).

THEOREM 11.1.1 (COMPARING *lsb* AND *lssb*).
Label-strong bisimulation is finer than label-semi-strong bisimulation.

PROOF. Clearly, a label-strong bisimulation is also a label-semi-strong bisimulation. The converse does not hold, as, e.g., the system in Figure 11.1 show. □

We have adopted *lsb* as the "right" notion of bisimulation, for it is global, and it is technically simpler. Furthermore, it is finer than *lssb*.

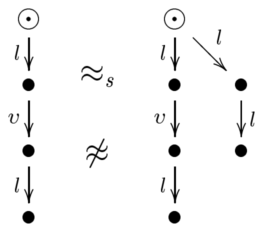

Figure 11.1: Lsb vs. lssb: comparing $l.\upsilon.l$ to $l.\upsilon.l + l.l$

11.1.2 Relation to other notions

What is the position of *lsb* in the lattice of bisimulation equivalences? Since it is a bisimulation it is above a large spectrum of equivalence notions [vG93]. Obviously, its relative position varies according to the characteristics of the transition system in consideration. We focus now on CCS and ABT.

As with weak bisimulation (*wb*), *lsb* is not a congruence in CCS. However, one defines from *lsb* a congruence (let us call it *lsc*) just by demanding that a silent action should be matched by at least one silent action (cf. the observational congruence, *oc*). Hence, *lsc* is finer than *oc* (as *lsb* is finer than *wb*), since the laws of *lsc* are particular cases of the laws of *oc*. In CCS, the coarsest bisimulation which is still a congruence is the progressing bisimulation (*pb*) [MS92]. Notice that *lsc* is incomparable to *pb*, as, e.g., $l.\upsilon.m \neq_{pb} l.\upsilon.\upsilon.m$ but $l.\upsilon.m \approx_{lsc} l.\upsilon.\upsilon.m$, and $l + \tau.l =_{pb} \tau.l$ but $l + \tau.l \not\approx_{lsc} \tau.l$.

In ABT, *wb* is a congruence, as the sums are prefixed. Since this setting has no mixed sums, the υ-laws are particular cases of the laws holding for *wb*. Thus, *wb* is still coarser than *lsb*, but notice that *pb* is, in this setting, finer than the previous two, since it distinguishes, e.g., $l.\upsilon.m$ from $l.\upsilon.\upsilon.m$ (hence, the law U1 — valid for *lsb* and for weak bisimulation — is not valid for *pb*).

11.2 Comparison with related work on types

Concurrency theory inspires dynamic notions of typing and subtyping. These dynamic notions of types, capturing some aspects of behaviour, have (at least) two different forms: *types and effects* and *regular types*.

In the following subsections we briefly present each approach and compare them to ABT.

11.2.1 Types and effects

The *type and effect discipline* is a framework for principal typing reconstruction in implicitly typed polymorphic functional languages [NN96, TJ94]. Types describe what expressions compute (sets of values) and effects describe how expressions compute (behaviour). In a concurrent functional scenario this discipline is particularly useful to prove 'subject reduction' and related results. Types and effects may decrease with computation. As effects (also called *behaviours*) model communication, their decrease corresponds to consuming prefixes, which suggests an operational semantics, and makes behaviours look like a process algebra, which is an abstraction of the semantics of the concurrent functional language.

In the context of name-passing calculi, types are assigned to names, not to processes, and are the semantics of the usage of names. Hence, we merge types and behaviours in a single entity.

11.2.2 Regular types

Behavioural typing and behavioural subtyping are notions of, respectively, typing and subtyping for concurrent object-oriented programming, which take into account dynamic aspects of objects' behaviour. In the beginning of the 90's, Nierstrasz argued that typing concurrent objects poses particular problems due to the non-uniform methods availability, since, by synchronisation constraints, the availability of a method depends upon the internal state of the object (which reflects the state of the system) [Nie95]. Therefore, a static notion of typing, like interfaces-as-types, is not powerful enough to capture dynamic properties of the behaviour of concurrent objects. Hence, Nierstrasz proposed the use of a regular language as types for active objects, to characterise the traces of menus offered by the objects. He also proposed a notion of subtyping, *request substitutability*, which is based on a generalisation of the *principle of substitutability* by Wegner and Zdonick [WZ88], according to the extension relation of Brinksma *et al.* [BSS87]. It is a transition relation, close to the failures model.

Nierstrasz's work was a strong inspiration. It provided us with the labelled sum as a base type. Since a notion of type equivalence is fundamental in our work we concentrate on its algebraic characterisation, leaving subtyping for subsequent work.

Several researchers are working on this track, further developing behavioural notions of typing and subtyping. We give here a brief account of their work, as a more thorough comparison to our work is in the conclusion of the thesis.

1. Bowman *et al.* define behavioural subtyping in LOTOS using its pre-orders and reinterpreting reduction [BBSDS97].
2. Puntigam [Pun96, Pun97, Pun99] defines a process algebra of types (with the expressiveness of a non-regular language), which enforces sequencing of messages. Subtyping is a central issue.

3. Najm and Nimour [NN97] develop types as sets of deterministic guarded parametric equations, equipped with a transition relation, which represent infinite state systems. The types are shaped by their calculus. They define an equivalence relation, a compatibility relation, and a subtyping relation on types, based on the simulation and on the bisimulation relations, and from there they build the type system.

11.3 Further work

The first priority is to find out if ABT is completely axiomatisable. From there, apart from the decision procedure for *lsb*, two topics are interesting: a modal characterisation, to specify properties, and a notion of subtyping, to allow program refinement.

11.3.1 Modal characterisation

To specify/verify properties of types it is useful to have a logical characterisation of the equivalence notion. In the process algebra realm this is done with a modal action logic like the Hennessy-Milner Logic [HM85, Mil89a]. In the same way we define a modal logic for ABT.

Definition 11.3.1 (Syntax).
The grammar below defines the set $\mathcal{F}$ of formulae *of the logic.*

$$\varphi ::= \top \mid \neg\varphi \mid \varphi \wedge \varphi \mid \langle v\rangle\varphi \mid \langle l(\widetilde{\alpha})\rangle\varphi$$

The relation below defines when does a type satisfy a formula.

Definition 11.3.2 (Semantics).
The following rules inductively define the satisfaction relation $\models\, \subseteq \mathcal{T} \times \mathcal{F}$.

1. $\alpha \models \top$, *for any* α;
2. $\alpha \models \neg\varphi$, *if not* $\alpha \models \varphi$;
3. $\alpha \models \varphi \wedge \psi$, *if* $\alpha \models \varphi$ *and* $\alpha \models \psi$;
4. $\alpha \models \langle v\rangle\varphi$, *if* $\exists_{\alpha'}(\alpha \Longrightarrow \alpha'$ *and* $\alpha' \models \varphi)$;
5. $\alpha \models \langle l(\widetilde{\alpha})\rangle\varphi$, *if* $\exists_{\alpha'}(\alpha \xrightarrow{l(\widetilde{\alpha})} \alpha'$ *and* $\alpha' \models \varphi)$.

An equivalence relation rises naturally from the satisfaction relation.

Definition 11.3.3 (Logical equivalence).
Types α and β are logically equivalent, $\alpha =_{lg} \beta$, *if, for all φ, we have $\alpha \models \varphi$, if, and only if, $\beta \models \varphi$.*

Logical equivalence is sound with respect to *lsb*.

THEOREM 11.3.1 (SOUNDNESS).
If $\alpha =_{lg} \beta$ *then* $\alpha \approx \beta$.

PROOF. By induction on the structure of the formulas. □

The converse direction is a conjecture. Usually, it requires image-finiteness, but we think that it is not the case here.

11.3.2 SUBTYPING

The notion we propose is a semantic notion of subtyping. A syntactic notion comes from extra rules of the type system, as for instance, in [PS96]. It is interesting to define those rules and study the relationship among both notions, and we expect them to coincide.

Furthermore, a decision procedure for subtyping is necessary in order to obtain a type inference algorithm.

Part IV

A type system for non-uniformity

Chapter 12

Motivation

To ensure that a process is error-free (Definition 5.1.7 in page 41), the interactions within the process should be disciplined in such a way that persistent bad redexes never appear. However, as mentioned in Section 5.2.1, checking whether a given process is error-free constitutes an undecidable problem. A popular approach to avoid run-time errors is the use of a static type discipline. Obviously, it can not be complete, but it ensures run-time safety, because type systems reject all programs that may lead to errors.

In sequential settings, like the λ-calculus, the types are functional, and constitute partial specifications of a term. In concurrent settings, like name-passing process calculi, a type system imposes a communication discipline, which prevents execution errors by assigning types to the names of a process. The types are partial specifications of the behaviour of names in the process: they specify the interactions that can happen on a name and the types of the names carried by that name. Hence, the types provide the necessary information to prevent communication errors. The purpose of the type-checking algorithm is to check whether a given set of type-assignment formulae (name-type pairs), called a typing, is adequate for a given process, i.e. whether it guarantees type safety. The algorithm ensures that no typing is adequate to an error process.

In this chapter we introduce the necessary terminology and notation to define the type system, and, with the help of some examples, we motivate the key issues of the type system. Chapter 13 defines two co-inductive relations which capture the two kinds of errors: *message-never-understood* and *blocked objects*. Chapter 14 presents the type system and its main properties.

12.1 The key ideas

We start with some examples to explain how the types represent the semantics of the usage of names in a process. For each construct of the type algebra, we give an example of an object whose behaviour is characterised by the construct.

The presentation of the examples requires some terminology, which we introduce below.

12.1.1 Typings

The type system assigns typings to processes. All the names of a process, except those that are parameters of methods, occur in the typing. We need to distinguish the usage of names within a process — either as locations of objects or as targets of messages — since it is the simultaneous presence of both in a process that may result in communication errors. Therefore, a type-assignment formula is a pair with a name as first component and a pair of types as second component: the first element of this pair is the type that characterises the input behaviour of the name, the second characterises the output behaviour of the name. This output behaviour is characterisable by terms of a grammar simpler than ABT.[1] However, defining two syntactic categories — input and output — would imply studying again axiomatic systems for *lsb*. The main flaw of working with two syntactic categories is that, since there are no labelled sums in output-types, the respective axiomatic system would not have an expansion law, and would be more complex than that in Definition 9.4.3, page 67. Therefore, although we refer to the two syntactic categories, it is only for the sake of clarity of language, since both correspond to the types in Definition 9.1.1, page 64.

Definition 12.1.1 (Typing judgements).

1. Type-assignment to names *are formulae* $a{:}(\alpha, \beta)$ *where* α *and* β *are the types of the name* a *as, respectively, subject of objects and subject of messages.*
2. Typings*,* Γ, Δ*, are finite sets of type-assignment formulae where no name occurs twice.*
3. Type sequence-assignment to process variables *are formulae* $X{:}(\widetilde{\alpha}, \widetilde{\beta})$ *where* $\widetilde{\alpha}$ *and* $\widetilde{\beta}$ *are the sequences of types of, respectively, the input and the output parameters of* X*.*
4. Bases*,* B*, are finite sets of type sequence-assignment formulae where no process variable occurs twice.*
5. Judgements *are formulae* $B, \Gamma \vdash P$*. If* $B = \emptyset$*, we write* $\Gamma \vdash P$ *and say that* P *is typable under* Γ.

[1] A possible grammar for output-types is the following:

$$\beta ::= \mathbf{0} \;\mid\; l(\beta) \;\mid\; \sum_{i\in I} \upsilon.\beta_i \;\mid\; \beta \parallel \beta \;\mid\; t \;\mid\; \mu t.\sum_{i\in I} \upsilon.\beta_i$$

12.1.2 EXAMPLES

By means of some basic examples, we show how to make use of the main constructs of the type algebra to build types suitable for the particular usage of a name in a process.

Recall that the objects are non-uniform — hence, with dynamic interfaces — and that we "implement" distributed objects by parallel composition of several objects with the same subject. The types reflect these facts.

EXAMPLE 12.1.2 (TYPING OBJECTS).

Interface types. Labelled sums represent objects with dynamic interfaces, like in $\{a{:}(l+m, \mathbf{0})\} \vdash a?\{l, m\}$, and in $\{a{:}(l.m, \mathbf{0})\} \vdash a?\{l = a?\{m\}\}$.

Notice that, due to the locality condition, the types of the parameters of methods are output-types, as in $\{a{:}(l(m), \mathbf{0})\} \vdash a?\{l(x) = x!m\}$.

Interleaving types. Parallel composition represent the behaviour of distributed objects: $\{a{:}(l \parallel m, \mathbf{0})\} \vdash a?\{l\} \mid a?\{m\}$; a parallel composition of messages is equally typed with the parallel composition operator of types: $\{a{:}(\mathbf{0}, l \parallel m)\} \vdash a!l \mid a!m$.

Notice that both processes are also typable with $l.m + m.l$.

Blocked types. Blocked sums represent blocked objects or blocked messages, like in $\{b{:}(m+n, \mathbf{0}), a{:}(v.l + v.m, \mathbf{0})\} \vdash b?\{m = a?\{l\}, n = a?\{m\}\}$, or in

$\{b{:}(m+n, \mathbf{0}), a{:}(\mathbf{0}, v.l + v.m)\} \vdash b?\{m = a!l, n = a!m\}$.

Non-deterministic sums of types. Recall the non-deterministic choice operator defined in Section 3.1.2. An object with internal non-determinism is typable with a blocked sum: $\{a{:}(l.(v.m + v.n), \mathbf{0})\} \vdash a?\{l = a?\{m\} \oplus a?\{n\}\}$.

Recursive types. Recursive types represent the behaviour of persistent objects, as in $\{a{:}(\mu t.l.(t \parallel m), \mathbf{0})\} \vdash (\mathsf{rec}\, X(x;).x?\{l = X[x;] \mid x?\{m\}\})[a;]$, or in

$\{a{:}(\mu t.l.(t \parallel \mu t'.m.t'), \mathbf{0})\} \vdash$
$(\mathsf{rec}\, X(xz;).x?\{l = X[xz;] \mid (\mathsf{rec}\, Y(y;).y?\{m = Y[y;]\})[z;]\})[aa;]$.

A message in the body of a method of a persistent object also behaves recursively:

$\{a{:}(\mu t.l.t, \mu t.v.(l \parallel t))\} \vdash (\mathsf{rec}\, Y(y;).y?\{l = Y[y;] \mid y!l\})[a;]$.

Furthermore, the output type is blocked, as the message is within the body of the method; notice that the types support auto-concurrency: an l message with the subject equal to the object, like $a!l[a]$, has output-type $\mu t.l(t)$.

One-place buffer type. Recall the one-place buffer of Example 5.1.4 in page 39; its type clearly specifies its behaviour: $\{b{:}(\mu t.(\mathit{write}(\alpha).\mathit{read}(\mathit{val}(\alpha)).t), \mathbf{0})\} \vdash \mathsf{Empty}[b]$, for any type α.

12.2 Terminology and notation

To proceed with the formal presentation of the type system we introduce further terminology and notation.

NOTATION. Let $\mathrm{dom}(\Gamma)$ be the set of the names in Γ. Consider a type-pair $\gamma \stackrel{\text{def}}{=} (\alpha, \beta)$, and let $\vec{\gamma} \stackrel{\text{def}}{=} (\widetilde{\alpha}, \widetilde{\beta})$ denote a sequence of type-pairs. Then:

1. $\Gamma \cdot a{:}\gamma$ denotes $\Gamma \cup \{a{:}\gamma\}$ provided that $a \notin \mathrm{dom}(\Gamma)$, and $\Gamma \setminus a$ denotes Γ without the formula with the name a.
2. Let $\Gamma(a) = \gamma$, if $a{:}\gamma \in \Gamma$, and $\Gamma(a) = (\mathbf{0}, \mathbf{0})$ otherwise.
3. If $\Gamma(a) = (\alpha, \beta)$, then $\Gamma(a)_{\mathsf{i}} = \alpha$ and $\Gamma(a)_{\mathsf{o}} = \beta$.
4. Let $\tilde{x}{:}(\widetilde{\alpha}, \widetilde{\beta})$ be a sequence of typing assignments $x_1{:}(\alpha_1, \beta_1), \ldots, x_n{:}(\alpha_n, \beta_n)$, for some n; consider $\Gamma \cdot \tilde{a}{:}\vec{\gamma}, \Gamma(\tilde{a}), \Gamma(\tilde{a})_{\mathsf{i}}$, and $\Gamma(\tilde{a})_{\mathsf{o}}$ to be the extensions of the definitions above;
5. Consider bisimulation and subtyping on γ-types pairwise defined; hence, assume that
   (a) $\Delta(a) \leq \Gamma(a)$, if $\Delta(a)_{\mathsf{i}} \leq \Gamma(a)_{\mathsf{i}}$ and $\Delta(a)_{\mathsf{o}} \leq \Gamma(a)_{\mathsf{o}}$, and that
   (b) $\Gamma' \approx \Gamma$, if, for all $a \in \mathrm{dom}(\Gamma) \cap \mathrm{dom}(\Gamma')$, $\Gamma(a)_{\mathsf{i}} \approx \Gamma'(a)_{\mathsf{i}}$ and $\Gamma(a)_{\mathsf{o}} \approx \Gamma'(a)_{\mathsf{o}}$;
6. For bases, we define $\mathrm{dom}(B)$ and $B{\cdot}X{:}\vec{\gamma}$ likewise the respective definitions for typings.

It is necessary to substitute names for other names in typings.

DEFINITION 12.2.1 (SUBSTITUTION ON TYPINGS).
A typing substitution*, denoted by* $\Gamma[z/x]$*, is defined as follows:* $(\Gamma \cdot x{:}\gamma)[z/x] = \Gamma \cup \{z{:}\gamma\}$ *if* $z \notin \mathrm{dom}(\Gamma)$ *or* $\Gamma(z) \approx \gamma$*, and* $\Gamma[z/x] = \Gamma$ *if* $x \notin \mathrm{dom}(\Gamma)$*.*

Chapter 13

Capturing the notion of error

Since a term of the type algebra defines the semantics of the usage of a name in a process, we refer to two syntactic categories of types — input and output — as a name may be an object's location or a message's target. An input-type characterises the sequences of messages that an object can accept, and thus the object life-cycle; an output-type characterises the messages sent to an object. Due to the locality condition (cf. the remark in Section 3.1.1, page 17), the names that are parameters of methods have an empty input-type and a non-empty output-type.

According to Definition 5.1.7 in page 41, an error-free process does not have persistent bad redexes. As pointed out in that section, errors are due to two situations: blockings on objects, or message-never-understood requests. Let us initially concentrate on the first. A process does not have a blocking, if all messages to its objects will eventually be accepted. Therefore, there must be a particular sequencing of the messages that allows their reception.

Think of types as graphs, or as rational trees (cf. [RV97]); an object accepts a set of messages if, either its output-type shares a path with its input-type, or a path of one is a prefix of a path of the other. However, as the calculus is non-deterministic, only when no sequencing of messages lead to blockings can one guarantee that the object accepts all the messages. Therefore, again in terms of graphs, all paths that have a common prefix must be equal.

Since the types are terms of the process algebra ABT, relations on types, based on labelled transitions, are an adequate form of coping with error-detection. We need two different ones, one to detect blockings, the other to detect messages-never-understood. It turns out that the latter is a generalisation of the former.

13.1 The intuition

To illustrate these ideas we discuss some examples explaining in detail each of the key requirements.

13.1.1 Detecting blockings

A relation on types, called *agreement on types*, detects blockings. Recall that the subject of the blocking is always a bound name. The relation "compares" input-types to output-types.

Example 13.1.1 (Agreement relation on types). Two types agree, if the following conditions hold.

1. *They share at least one path.*

   Take the process $P \stackrel{\text{def}}{=} a?\{l = a?\{m\}\}$ where the input-type of the name a is $l.m$; process $\nu a\, P \,|\, a!m$ has a blocking, as the object cannot change its state to accept the message. In terms of types, the output-type m does not match the first transition of the input-type $l.m$. However, the output-type $m \parallel l \parallel n$, corresponding to the output usage of a in $Q \stackrel{\text{def}}{=} \nu a\, P \mid a!m \mid a!l \mid a!n$, matches the transition $l.m$, thus ensuring statically the absence of blockings on a.

   Further notice that $Q \xrightarrow{\tau}\xrightarrow{\tau} \nu a\, a!n$, which is obviously not an error, hence the agreement relation holds when one of the types is empty.

2. *Paths sharing prefixes match.*

   Take the process $P \stackrel{\text{def}}{=} a?\{l = a?\{m\}, m = a?\{n\}\}$ where the input-type of the name a is $l.m + m.n$; process $\nu a\, P \mid a!m \mid a!l$ has a potential blocking that happens if a accepts first the m-message. In terms of types, the output-type $m \parallel l \parallel n$ matches the transition $l.m$, but afterwards, the type m fails to match the transition n. Thus, the agreement relation does not rely only on the existence of a common path — it additionally requires that paths sharing prefixes should match.

Regarding the relation, there are two important aspects to be aware of.

1. *It is a higher-order relation.*

   The relation only holds when the types that are parameters of labels are also in the relation. Thus, the relation must capture some causal dependency on the usage of names, as explained in the following example. Take the process $a?\{l(x) = x!m\} \mid b?\{m = b?\{n\}\} \mid a!l[b] \mid b!n$; a typing for it is $\{a{:}(l(m), l(m \parallel n)), b{:}(m.n, m \parallel n)\}$. Notice that input-type of a agrees with its output-type, since both have a transition l, and the output-type of b — the parameter of the output-type of a — also agrees with its input-type.

2. *When can an unblocking transition take place?*

   An unblocking transition corresponds to a method execution. Dealing with these transitions in the agreement relation is a delicate matter. To guarantee the safety of programs, unblocking at a type of a given name can only occur if the process does not have a bad redex with that name as subject. Take the process $\nu a\, a?\{l = b?\{m\} \mid b!m\} \mid a!l$; a possible type for b is $(\upsilon.m, \upsilon.m)$, and as there are no bad redexes, the unblocking transitions can safely occur. Take now $P \stackrel{\text{def}}{=} \nu b\, a?\{l = b!m\} \mid b?\{m = b?\{l\}\} \mid b!l$ and $Q \stackrel{\text{def}}{=} \nu a\, P$. Notice that P is not an error, but Q is, since in P the bad redex is not necessarily persistent,[1] but it is so in Q. Nevertheless, both processes are typable by a typing where the type of b is $(m.l, \upsilon.m \parallel l)$. The interface of the input-type is different from that of the output type, thus both P and Q have a bad bl-redex. For the sake of safety, in this situation unblockings are disallowed.

   Therefore, release transitions occur only when the interfaces of both types are equal. Notice that divergent error-processes are detected:

   $\Gamma \stackrel{\text{def}}{=} \{a{:}(\mu t.m.t, \mu t.\upsilon.m.t \parallel l \parallel m)\} \vdash (\mathsf{rec}\, X(x;).x?\{m = X[x] \mid x!m\})[a;] | a!m | a!l$ holds, but $\Gamma(a)_{\mathsf{i}}$ and $\Gamma(a)_{\mathsf{o}}$ do not agree.

13.1.2 DETECTING MESSAGE-NEVER-UNDERSTOOD REQUESTS

An object with a free subject not under prefixes is never blocked, as it can always receive messages from the environment. Furthermore, only a persistent object may participate in an error, as ephemeral ones can be consumed by receiving messages (see Example 5.1.4.2 and Example 5.1.4.4 in page 39).

Therefore, a transition relation on types designed to detect message-never-understood requests, called *compatibility of types*, holds if the output-type is empty (no message, no redex), if the input-type is finite (no persistent object, no persistent redex), or if, with some "help" from the environment, providing the messages that correspond to the missing transitions in the input type, the input-type agrees with the output-type.

13.2 TWO NEW RELATIONS ON TYPES

We are now in a position to formally define the relations used to detect error-processes: one capturing blockings, the other message-never-understood requests. The former is the *agreement* relation, the latter the *compatibility* relation, both co-inductive higher-order contravariant relations on types.

[1] $P \xrightarrow{a?l} \xrightarrow{\tau} \xrightarrow{\tau} \mathbf{0}$, all transitions being deterministic.

The agreement relation ensures the absence of blockings: an output-type agrees with an input-type, if they share at least one path; and all paths which have a common prefix are equal up-to agreement on the parameters, contravariantly. Unblocking occurs only when at least one of the types is blocked, or when both types have the same interface — a safety approach to detect bad redexes. Moreover, if one of the types is empty then the relation always holds, regardless of what the other type is (a message or an object by themselves are not errors).

13.2.1 Agreement relation

We present now the definition of the relation on types that captures errors due to blocked objects. It requires the notions of interface of a type in Definition 8.1.2, page 56.

DEFINITION 13.2.1 (AGREEMENT ON TYPES). *Fix Γ and P.*

1. *A binary relation $S \subseteq \mathcal{T} \times \mathcal{T}$ is an* agreement on Γ and P, *if whenever $\alpha \; S \; \beta$,*
   (a) *$\beta \approx \mathbf{0}$ or $\alpha \approx \mathbf{0}$, or else*
   (b) i. *$\exists_{l,\alpha',\beta',\widetilde{\beta}_1,\widetilde{\beta}_2} \, \alpha \xrightarrow{l(\widetilde{\beta}_1)} \alpha'$ and $\beta \xrightarrow{l(\widetilde{\beta}_2)} \beta'$ and $\widetilde{\alpha}_2 \; S \; \widetilde{\beta}_1$ for all $a!l[\tilde{v}]$ subterms of P such that $\Gamma(a) \approx (\alpha, \beta)$ and $\Gamma(\tilde{v}) \approx (\widetilde{\alpha}_2, \widetilde{\beta}_2)$, and $\alpha' S \beta'$,*
   ii. *$\forall_{l,\alpha',\beta',\widetilde{\beta}_1\widetilde{\beta}_2} (\alpha \xrightarrow{l(\widetilde{\beta}_1)} \alpha'$ and $\beta \xrightarrow{l(\widetilde{\beta}_2)} \beta'$ and $\widetilde{\alpha}_2 \; S \; \widetilde{\beta}_1$ for all $a!l[\tilde{v}]$ subterms of P such that $\Gamma(a) \approx (\alpha, \beta)$ and $\Gamma(\tilde{v}) \approx (\widetilde{\alpha}_2, \widetilde{\beta}_2))$ implies $\alpha' S \beta'$, and*
   iii. *if $\mathrm{int}(\alpha) = \emptyset$ or $\mathrm{int}(\beta) = \emptyset$ or $\mathrm{int}(\alpha) = \mathrm{int}(\beta)$, then*
   $\alpha \xrightarrow{v} \alpha'$ implies $\exists_{\beta'} (\beta \Longrightarrow \beta'$ and $\alpha' \; S \; \beta')$, and
   $\beta \xrightarrow{v} \beta'$ implies $\exists_{\alpha'} (\alpha \Longrightarrow \alpha'$ and $\alpha' \; S \; \beta')$.
2. *A type α* agrees *with a type β on a typing Γ and a process P, written $\alpha \bowtie_\Gamma^P \beta$, if there is an agreement relation S on Γ and P such that $\alpha \; S \; \beta$.*

Observe that for arbitrary Γ and P the notion looks awkward. However, we only use it when $\Gamma \vdash P$, cf. Definition 14.1.2 ahead.

PROPOSITION 13.2.1.
The relation $\bowtie_\Gamma^P$ is reflexive and symmetric.

PROOF. Directly from the definition. $\square$

REMARK. Notice that the relation $\bowtie_\Gamma^P$ is:

1. not transitive: $l \bowtie_\Gamma^P (l \parallel m) \bowtie_\Gamma^P m$ holds, but $l \not\bowtie_\Gamma^P m$;
2. not a congruence, as it is not preserved by parallel composition: $(l+m) \bowtie_\Gamma^P m$ holds, but $((l+m) \parallel l) \not\bowtie_\Gamma^P (m \parallel l)$, since condition 1.(b)ii of the definition above fails.

13.2.2 COMPATIBILITY OF TYPES

The compatibility relation ensures the absence of messages-never-understood. It is essentially a relaxing of the previous relation by allowing the environment to "help", providing some steps: an output-type is compatible with an input-type if the former has a path that coincides with a *projection* of a path of the latter.

We define likewise to the agreement relation the *type compatibility relation* $\prec_\Gamma^P$, replacing '$\alpha \equiv \mathbf{0}$' with 'α finite' in the condition 1.(a) of Definition 13.2.1, and '$\alpha \xrightarrow{l(\widetilde{\beta}_1)} \alpha'$' with '$\alpha \xrightarrow{\widetilde{\pi}} \alpha \xrightarrow{l(\widetilde{\beta}_1)} \alpha'$, for some $\widetilde{\pi}$ without occurrences of silent transitions', in conditions 1.(b)i and 1.(b)ii.

DEFINITION 13.2.2 (COMPATIBILITY OF TYPES). *Fix Γ and P.*

1. *A binary relation $S \subseteq \mathcal{T} \times \mathcal{T}$ is a* compatibility on Γ and P, *if whenever $\alpha\, S\, \beta$,*
   (a) *$\beta \approx \mathbf{0}$ or α is finite,*[2] *or else*
   (b) i. *$\exists_{l,\alpha',\beta',\widetilde{\beta}_1,\widetilde{\beta}_2}\, \alpha \xrightarrow{\widetilde{\pi}} \alpha \xrightarrow{l(\widetilde{\beta}_1)} \alpha'$, for some $\widetilde{\pi}$ without occurrences of silent transitions, and $\beta \xrightarrow{l(\widetilde{\beta}_2)} \beta'$ and $\widetilde{\alpha}_2\, S\, \widetilde{\beta}_1$ for all $a!l[\tilde{v}]$ subterms of P such that $\Gamma(a) \approx (\alpha, \beta)$ and $\Gamma(\tilde{v}) \approx (\widetilde{\alpha}_2, \widetilde{\beta}_2)$, and $\alpha' S \beta'$,*
   ii. *$\forall_{l,\alpha',\beta',\widetilde{\beta}_1\widetilde{\beta}_2}\, (\alpha \xrightarrow{\widetilde{\pi}} \alpha \xrightarrow{l(\widetilde{\beta}_1)} \alpha'$, for some $\widetilde{\pi}$ without occurrences of silent transitions, and $\beta \xrightarrow{l(\widetilde{\beta}_2)} \beta'$ and $\widetilde{\alpha}_2\, S\, \widetilde{\beta}_1$ for all $a!l[\tilde{v}]$ subterms of P such that $\Gamma(a) \approx (\alpha, \beta)$ and $\Gamma(\tilde{v}) \approx (\widetilde{\alpha}_2, \widetilde{\beta}_2))$ implies $\alpha' S \beta'$, and*
   iii. *if $\mathrm{int}(\alpha) = \emptyset$ or $\mathrm{int}(\beta) = \emptyset$ or $\mathrm{int}(\alpha) = \mathrm{int}(\beta)$ then*
   $\alpha \xrightarrow{v} \alpha'$ implies $\exists_{\beta'}(\beta \Longrightarrow \beta'$ and $\alpha'\, S\, \beta')$, and
   $\beta \xrightarrow{v} \beta'$ implies $\exists_{\alpha'}(\alpha \Longrightarrow \alpha'$ and $\alpha'\, S\, \beta')$.
2. *A type α is* compatible *with a type β on a typing Γ and a process P, written $\alpha \prec_\Gamma^P \beta$, if there is a compatibility relation S on Γ and P such that $\alpha\, S\, \beta$.*

The relation $\prec_\Gamma^P$ is reflexive, but it is neither symmetric nor transitive, and further, it is not a congruence. Proofs and counter-examples for these results are similar to the respective ones in the previous section.

PROPOSITION 13.2.2.
The relation $\prec_\Gamma^P$ is reflexive.

PROOF. Directly from the definition. □

Notice that *lsb* preserves both the agreement and the compatibility relations.

[2]cf. Definition 8.1.1 on page 56

Proposition 13.2.3.
Consider $\alpha \approx \alpha'$ and $\beta \approx \beta'$.

1. *If $\alpha \bowtie_\Gamma^P \beta$ then $\alpha' \bowtie_\Gamma^P \beta'$.*
2. *If $\alpha \prec_\Gamma^P \beta$ then $\alpha' \prec_\Gamma^P \beta'$.*

Proof. Simply apply the definitions. $\square$

Lemma 13.2.4 (Compatibility).
For all $a \in \mathrm{dom}(\Gamma)$, $\Gamma(a)_i \prec_\Gamma^{P|a!l[\tilde{v}]} \Gamma(a)_o$ implies $\Gamma(a)_i \prec_\Gamma^P \Gamma(a)_o$.

Proof. Directly from the definition. $\square$

Chapter 14

Type checking system

We are finally in a position to present the type checking system, and prove that it is correct with respect to the notion of error-free process.

The rules are fairly simple. They check that:

1. the types of the arguments of a recursive call are compatible with the types of their respective parameters — ensuring that the calculus is indeed local;
2. the types of messages in recursive objects are recursive blocked sums;
3. the labels of messages and objects appear in nodes of the rational tree that its type denotes.

The type system *per se* does not guarantee the absence of run-time errors in typable processes, i.e. that the type system is sound. After checking the basic conditions we have just described, it is necessary to "calculate" a minimal typing, that is a typing with only the strictly necessary pairs, and with the simplest possible types. After doing so, we should still verify that, for all names in the minimal typing of the process, the types of the free output-names are compatible with the types of the free input-names, and that the types of the bound output-names agree with the types of the bound input-names. Only when these conditions are fulfilled the system ensures that no typable process will attain a run-time error.

We start by presenting the system, then introduce and define minimal typings, and finally, prove the main results — the correctness of the type checking system with respect to the notion of error, and the uniqueness of types up to type bisimulation.

14.1 The type system

The type checking system possesses a type rule for each process constructor, and an extra rule (called $\approx$) allows the substitution in a typing formula of a type with a bisimilar one. Thus, when type-checking a process, there are always two rules that may be used: that of the constructor under analysis, or $\approx$. The latter should only be applied to enable the use of that corresponding to the constructor.

14.1.1 Rules

The type systems requires the notions of type equivalence (Definition 7.3.1, page 50), of subtyping (Definition 9.3.1, page 66), and of restriction of a typing to a set of names, which is below.

DEFINITION 14.1.1 (RESTRICTION OF TYPINGS).
The restriction *of a typing* Γ *to the set of names* N*, denoted* $\Gamma \restriction N$*, is the following typing:* $\{x{:}\Gamma(x) \mid\ x \in N \cap \mathrm{dom}(\Gamma)\}$.

DEFINITION 14.1.2 (TYPE SYSTEM).
The rules of Table 14.1 inductively define the type checking system.

Some rules deserve further explanation:

- rule VAR checks that the types of the arguments of the process variable are equivalent to the types of its parameters, since in this case, the recursive call respects the "expectations" of the object definition;
- rule MSG checks that the output-type of the message's target may perform a transition labelled with the message's label (with the correct parameters), possibly after the occurrence of some unblockings;
- rule OBJ checks three conditions: (1) all the parameters of the methods are only used for output; (2) the input-type of the object a has a state that may perform a transition labelled with each of the names of the methods (with the correct parameters); and (3) the output-types of the output-names occurring in the object's methods offer upsilon transitions, and that the input-names different from a have a state that is a blocked sum (since the corresponding messages and objects are blocked);
- rule REC also checks two conditions: (1) the input-types of the arguments of the process variable have a state that is a subtype of the input-types of its parameters, and the output-types of its arguments are subtypes of the output-types of its parameters (since the arguments may have other occurrences than those defined by the persistent object); and (2) the output-types of the arguments are recursive blocked sums.

Notice that, in the rule RES, we do not discard the bound variable from the typing, because we may need its type to verify the agreement relation. Further notice that, to use the rule PAR, both processes should have the same typing, hence rule $\approx$ may have to be used.

Nil	$B, \Gamma \vdash \mathbf{0}$	
Var	$B \cdot X{:}\vec{\gamma}, \Gamma \cdot \vec{v}{:}\vec{\gamma} \vdash X[\vec{v}]$	
Msg	$B, \Gamma \vdash a!l[\tilde{v}]$	$(\Gamma(a)_{\mathsf{o}} \Longrightarrow\xrightarrow{l(\Gamma(\tilde{v})_{\mathsf{o}})})$
Obj	$\dfrac{\forall_{i\in I}\ B,\Gamma\cdot\tilde{x}_i{:}(\mathbf{0},\widetilde{\beta_i}) \vdash P_i}{B,\Gamma\vdash a?\{l_i(\tilde{x}_i)=P_i \mid i\in I\}}$	$\left(\begin{array}{l} \exists_{\tilde\pi,\alpha}\,(\Gamma(a)_{\mathsf{i}} \stackrel{\tilde\pi}{\Longrightarrow} \alpha \text{ and } \forall_{i\in I}\,\alpha \xrightarrow{l_i(\widetilde{\beta_i})}) \text{ and}\\ \forall_{b\in\mathrm{dom}(\Gamma\restriction\cup_{i\in I}\mathrm{n}(P_i))}\,(\Gamma(b)_{\mathsf{o}} \stackrel{\upsilon}{\Longrightarrow} \text{ and if } b\neq a \text{ then}\\ \quad \exists_{\tilde\pi,\alpha}\,(\Gamma(b)_{\mathsf{i}} \stackrel{\tilde\pi}{\Longrightarrow}\alpha \text{ and } \forall_{i\in I}\exists_{\alpha_i}\,\alpha \stackrel{\upsilon}{\Longrightarrow}\alpha_i))\end{array}\right)$
Rec	$\dfrac{B\cdot X{:}\vec\gamma,\Gamma\cdot\vec{x}{:}\vec\gamma\vdash a?M}{B,\Gamma\cdot\vec{v}{:}\vec\gamma'\vdash(\mathbf{rec}\,X(\vec{x}).a?M)[\vec{v}]}$	$\left(\begin{array}{l}\exists_{\tilde\pi}\,\vec\gamma'_{\mathsf{i}}\stackrel{\tilde\pi}{\Longrightarrow}\le\vec\gamma_{\mathsf{i}},\ \vec\gamma'_{\mathsf{o}}\le\vec\gamma_{\mathsf{o}}, \text{ and}\\ \forall_{\gamma\in\{\vec\gamma_{\mathsf{o}}\}}\gamma\approx\mu t.\sum_{i\in I}\upsilon.\beta_i\end{array}\right)$
Res	$\dfrac{B,\Gamma\vdash P}{B,\Gamma\vdash \nu x\, P}$	
Par	$\dfrac{B,\Gamma\vdash P \quad B,\Gamma\vdash Q}{B,\Gamma\vdash P \mid Q}$	
$\approx$	$\dfrac{B,\Gamma\cdot x{:}\gamma\vdash P}{B,\Gamma\cdot x{:}\delta\vdash P}$	$(\gamma\approx\delta)$

Table 14.1: Type checking system.

14.1.2 Examples

The examples of Chapter 12, albeit simple, already illustrate the expressive power of the type algebra and how it is suitable for representing the behaviour of non-uniform objects. The previous chapter defines the transition relations that allow us to detect errors. The examples below emphasise the power and the limit of those relations.

EXAMPLE 14.1.3 (TYPING NON-UNIFORM OBJECTS).

Persistent objects. Consider the following three objects (cf. [CPS97]):

1. $(\mathsf{rec}\, X(a;).a?\{rep = X[a;\,] \mid a?\{mess\}\})[a;]$,
2. $(\mathsf{rec}\, X(b;).b?\{quest(x) = X[b;\,] \mid x!rep\})[b;]$, and
3. $(\mathsf{rec}\, X(c;).c?\{rep = X[c;\,], mess = X[c]\})[c;]$.

The parallel composition of all the objects together with the messages $b!quest[a]$ and $b!quest[c]$ is an error-free process. It is easy to check that:

1. $\Gamma(a)_{\mathsf{i}} = \mu t.rep.(t \| mess) \bowtie^P_\Gamma rep = \Gamma(a)_{\mathsf{o}}$,
2. $\Gamma(b)_{\mathsf{i}} = \mu t.quest(rep).t \bowtie^P_\Gamma quest(rep) \| quest(rep) = \Gamma(b)_{\mathsf{o}}$, and
3. $\Gamma(c)_{\mathsf{i}} = \mu t.rep.t + mess.t \bowtie^P_\Gamma rep = \Gamma(c)_{\mathsf{o}}$.

Blocked objects.

1. Let $\Gamma \stackrel{\text{def}}{=} \{b{:}(\mu t.write(\alpha).read(val(\alpha)).t, read(val(\alpha)))\}$ and $P \stackrel{\text{def}}{=} \nu b\, \mathsf{Empty}[b] \mid b!read[r]$. Observe that $\Gamma(b)_{\mathsf{i}} \not\bowtie^P_\Gamma \Gamma(b)_{\mathsf{o}}$ and P is an error, even though $\Gamma \vdash P$.
2. Let $\Gamma \stackrel{\text{def}}{=} \{a{:}(m, v.n), b{:}(l, \mathbf{0})\}$ and $P \stackrel{\text{def}}{=} \nu ab\, b?\{l = a!n\} \mid a?\{m\}$. Notice that $\Gamma(a)_{\mathsf{i}} \not\bowtie^P_\Gamma \Gamma(a)_{\mathsf{o}}$, whereas P is not an error, and $\Gamma \vdash P$.
3. Let $\Gamma \stackrel{\text{def}}{=} \{a{:}(n.m, v.n \parallel m), b{:}(l, l)\}$ and $P \stackrel{\text{def}}{=} \nu a\, b?\{l = a!n\} \mid a?\{n = a?\{m\}\} \mid a!m \mid b!l$. Albeit P is neither an error nor a deadlock, and moreover, $\Gamma \vdash P$, still $\Gamma(a)_{\mathsf{i}} \not\bowtie^P_\Gamma \Gamma(a)_{\mathsf{o}}$.

14.2 Minimal typings

The typing rules are general enough to type check more than one typing for a given process. This is due to three main reasons:

1. the names in the typing do not necessarily coincide with the free names in the process plus the parameters of methods;
2. the rule $\approx$ allows the substitution in a typing of a type for a bisimilar one;
3. the rule REC allows the substitution in a typing of a type for a supertype. Moreover, the side conditions of the rules MSG and OBJ can be fulfilled by many types, which are in the subtyping relation.

Rationale. It is simple to verify that the type system accepts more than one type-pair for each name in a process. There are three possible situations:

1. it accepts bisimilar types;
2. it accepts types that are in the subtyping relation; and
3. it accepts types that have more labels than those of the respective objects.

Take a process like $a?m \mid a?l$ and a typing Γ such that $\Gamma \vdash P$. Notice that possible input-types of a are $m \parallel l$, $l \parallel m$, $m.l + l.m$, and $l.m + m.l$, all bisimilar. Consider now $P \stackrel{\text{def}}{=} a?\{l = a?m\} \mid a!m$; a *minimal* typing of P is $\Gamma \stackrel{\text{def}}{=} \{a{:}(l.m, m)\}$, but there are subtypes of $\Gamma(a)$ which also succeed in type-checking P (like $(l.m, l \parallel m)$, even though there is no message l with subject a). Furthermore, the input rule allows that $\Gamma(a)_{\mathsf{i}}$ have labels that are not present in the object a (like $\Gamma(a)_{\mathsf{i}} = o.m.l$).

Moreover, an error process can type-check with a typing "too generous" for it: $\{a{:}(l.m, l \parallel m)\} \vdash \nu a\, P$, since $\Gamma(a)_{\mathsf{i}} \prec_{\Gamma}^{\nu a\, P} \Gamma(a)_{\mathsf{o}}$, even though $\nu a\, P$ is an error.

The definition. We need a notion of "minimal typing" to overcome undesirable situations such as those described above. Minimal typings gather three properties: (1) contain the smallest possible number of formulae; (2) do not have labels that are not present in the respective objects; and (3) the types occurring in the formulae are supertypes of the types occurring in the respective formulae of the other typings.

Definition 14.2.1 (Minimal typings).

1. *A typing* Γ *is* smaller *than a typing* Δ, $\Delta \leq \Gamma$, *if*
   (a) $\operatorname{dom}(\Delta) \subseteq \operatorname{dom}(\Gamma)$, *and*
   (b) $\Delta(a) \leq \Gamma(a)$, *for all* $a \in \operatorname{dom}(\Delta)$;
2. *A typing* Γ *is* acceptable *with respect to a process* P, *if*
   $P \xrightarrow{a?l[\tilde{v}]} Q$ *implies* $\Gamma(a)_i \Longrightarrow\xrightarrow{l(\tilde{\beta})} \alpha$ *and* $\Gamma \setminus a \cdot a{:}(\alpha, \Gamma(a)_o)$ *is acceptable with respect to* Q
3. *A typing* Γ is a minimal typing of a process P, *if*
   (a) $\Gamma \vdash P$;
   (b) Γ *is acceptable with respect to* P; *and*
   (c) $\forall_\Delta\, \Delta \vdash P$ *implies* $\Delta \leq \Gamma$.

14.3 Main results

We present here the main results regarding the type system, as well as sketches of their proofs. The main purpose of the type system is the detection of potential run-time errors.

Being an error is a property of processes which, in this case, cannot be compositionally checked. Thus, we cannot have a side condition on the rule PAR verifying whether the input types of all names in a typing are compatible or agree with their respective output types.

Therefore, after type checking a process, we still have to verify if its minimal typing ensures that the process is error-free, that is, to verify if the types of the free names are compatible, and if the types of the bound names agree.

DEFINITION 14.3.1 (WELL-TYPED PROCESSES).
A process P is well-typed *if it is typable and its minimal typing Γ satisfies the following conditions.*

1. *for all* $a \in \mathrm{dom}(\Gamma) \cap \mathrm{fn}(P)$, $\Gamma(a)_i \prec_\Gamma^P \Gamma(a)_o$*;*
2. *for all* $a \in \mathrm{dom}(\Gamma) \cap \mathrm{bn}(P)$, $\Gamma(a)_i \bowtie_\Gamma^P \Gamma(a)_o$.

In this case we say that Γ is a well-typing of P.

14.3.1 Results on typings

We adapt to our setting some standard auxiliary results that will be needed to prove Subject-Reduction and its corollary.

LEMMA 14.3.1 (FINITE ADDITIVITY).
Let A and B be sets of names. Then:

1. $\Gamma \upharpoonright (A \cup B) = \Gamma \upharpoonright A \cup \Gamma \upharpoonright B$*; and*
2. $(\Gamma \cup \Delta) \upharpoonright A = \Gamma \upharpoonright A \cup \Delta \upharpoonright A$.

PROOF. Simply by applying the definition. □

LEMMA 14.3.2 (BASIC PROPERTIES OF TYPINGS).

1. *(Free names) If* $B, \Gamma \vdash P$ *then* $\mathrm{fn}(P) \subseteq \mathrm{dom}(\Gamma)$ *and* $\mathrm{fv}(P) \subseteq \mathrm{dom}(B)$.
2. *(Narrowing) If* $\Gamma \vdash P$ *then* $\Gamma \upharpoonright \mathrm{n}(P) \vdash P$.
3. *(Weakening) If* $\Gamma \upharpoonright \mathrm{n}(P) \vdash P$ *then* $\Gamma \vdash P$.

PROOF. In all the three items, an induction on the derivation of $\Gamma \vdash P$ carries the proof.

1. (Free names) Straightforward.
2. (Narrowing) There are two base cases:
   (a) axiom Nil: the result follows trivially;
   (b) axiom Msg, the result follows by item 1 of this lemma, noticing that $\Gamma \upharpoonright (a!l[\tilde{v}])$ still fulfils the axiom's side-condition;

   For the induction step, the interesting cases are when the deduction ends with the rules Obj or Rec:
   (a) rule Obj: by induction hypothesis, it holds that $\forall_{i\in I}\,(\Gamma \cdot \tilde{x}_i{:}(\mathbf{0},\widetilde{\beta_i})) \upharpoonright \mathrm{n}(P_i) \vdash P_i$, hence, using Lemma 14.3.1, $\forall_{i\in I}\,\Gamma \upharpoonright \mathrm{n}(P_i) \cdot \tilde{x}_i{:}(\mathbf{0},\widetilde{\beta_i}) \upharpoonright \mathrm{n}(P_i) \vdash P_i$, and by weakening we finally get $\forall_{i\in I}\,\Gamma \upharpoonright \mathrm{n}(P_i) \cdot \tilde{x}_i{:}(\mathbf{0},\widetilde{\beta_i}) \vdash P_i$; the proof concludes by applying the rule Obj, noticing that the side condition holds by hypothesis;
   (b) rule Rec: Similar to the previous case.
3. (Weakening) Straightforward, by induction on the derivation of $\Gamma \upharpoonright \mathrm{n}(P) \vdash P$. Notice that only the names in $\mathrm{dom}(\Gamma) \cap \mathrm{n}(P)$ play a significant role in the typing rules.

□

Corollary 14.3.3.
If $\Gamma \vdash P$ and $\Gamma \subseteq \Delta$ then $\Delta \vdash P$.

Proof. Since $\Gamma \vdash P$ and $\Gamma \subseteq \Delta$ then $\Gamma \upharpoonright \mathrm{n}(P) = \Delta \upharpoonright \mathrm{n}(P)$, hence, by the narrowing lemma $\Delta \upharpoonright \mathrm{n}(P) \vdash P$ and the result follows applying the weakening lemma. □

Lemma 14.3.4 (Basic properties of typable processes).
Consider that $\Gamma \vdash P$.

1. *(Substitution) If $x \in \mathrm{fn}(P)$ and $\Gamma[z/x]$ is defined, then $\Gamma[z/x] \vdash P[z/x]$.*
2. *(Congruence) If $P \equiv Q$ then $\Delta \vdash Q$, for some Δ.*
3. *(Typability of subterms) If Q is a subterm of P, then $\Delta \vdash Q$, for some Δ.*

Proof.

1. (Substitution) We have to show that for each deduction of $\Gamma \vdash P$ there is a deduction of $\Gamma[z/x] \vdash P[z/x]$. We proceed by induction on the derivation of $\Gamma \vdash P$. Notice that, since $x \in \mathrm{fn}(P)$, by the free names' lemma (14.3.2.1), $x \in \mathrm{dom}(\Gamma)$; as $\Gamma[z/x]$ is defined by hypothesis, then one of the following two conditions hold:
   (a) either $z \notin \mathrm{dom}(\Gamma)$, so $z \notin \mathrm{fn}(P)$,
   (b) or $\Gamma \stackrel{\text{def}}{=} \Gamma' \cdot x{:}\gamma$, $\Gamma(z) \approx \gamma$, and $(\Gamma' \cdot x{:}\gamma)[z/x] = \Gamma' \cup \{z{:}\gamma\}$.

If it is the first condition that holds, the proof is straightforward. We assume now that it is the second condition that holds and proceed with the proof. There are two base cases to consider, depending on the axiom: if the deduction is the axiom NIL the result follows trivially; if it is the axiom MSG, since by hypothesis $\Gamma(z) \approx \gamma$, then the result follows by applying rule $\approx$.

For the induction step, the only interesting cases are when the deduction ends with rule OBJ or rule REC:

(a) rule OBJ: since $x \in \mathrm{fn}(a?M)$, if x is not a, the result follows using the induction hypothesis; otherwise, as by hypothesis $\Gamma(z) \approx \Gamma(a)$, the rule's side condition holds and so, using the induction hypothesis, the result follows again;

(b) rule REC: notice that, as $\mathrm{fn}(a?M) = \{\vec{x}\}$, then $(\Gamma \cdot \vec{x}{:}\vec{\gamma})[z/x] = \Gamma \cdot (\vec{x}{:}\vec{\gamma})[z/x]$; the result follows using the Proposition 9.3.2 and the induction hypothesis.

2. (Congruence) By a case analysis of the rules defining $\equiv$.

   (a) Alpha-conversion requires a proof by induction on the derivation of $\Gamma \vdash P$, using the substitution lemma. The only interesting cases are those involving binding operators.

      i. Rule OBJ; notice that, by the substitution lemma, $(\Gamma \cdot \tilde{x}{:}\widetilde{\gamma})[z/x] \vdash P_i[z/x]$, for all $i \in I$, and the result follows using the typing Γ/z.

      ii. Rule REC is treated similarly to the previous case.

      iii. Rule RES; simply notice that $\Gamma[y/x] \vdash (\nu x\, P)[y/x]$.

   (b) For the remaining rules, simple rearrangements on the derivation trees of $\Gamma \vdash P$ yield the result.

3. (Typability of subterms) Straightforward, by induction on the structure of P.

$\square$

14.3.2 RESULTS ON MINIMAL TYPINGS

As mentioned in the motivation to Definition 14.2.1, the typability of a process is a property that is weaker than we need. Candidates for "good" typings are those that are minimal. We characterise them herein, showing that typable processes have unique minimal typings, and that the recursive types of output names are blocked sums — an important fact to ensure the soundness result.

LEMMA 14.3.5. *If Γ is a minimal typing for P then $\Gamma = \Gamma \restriction \mathrm{n}(P)$.*

PROOF. The inclusion $\subseteq$ is obvious; the opposite results from narrowing and from Definition 14.2.1. $\square$

COROLLARY 14.3.6 (DOMAIN OF A MINIMAL TYPING).
Let $\Gamma \vdash P$ *and* Δ *be a minimal typing of* P. *Then,* $\mathrm{dom}(\Delta) \subseteq \mathrm{dom}(\Gamma \restriction \mathrm{n}(P))$.

PROOF. Follows by the narrowing lemma and the lemma above. □

PROPOSITION 14.3.7 (BASIC PROPERTIES OF MINIMAL TYPINGS).

1. *(Existence) A typable process has a minimal typing.*
2. *(Uniqueness) A minimal typing of a process is unique up to type equivalence.*

PROOF.

1. (Existence) Let $B, \Gamma \vdash P$. Then, by the narrowing lemma and by the above lemma, a minimal typing Δ of P does not have more names than those in P. We proceed by induction on the derivation of $B, \Gamma \vdash P$, showing that P is typable by a suitable candidate to minimal typing, and that this candidate is indeed a minimal typing of P. The bases cases are when the derivation is an axiom. The case of the axiom NIL is obvious: $\emptyset$ is the minimal typing of $\mathbf{0}$.
   (a) Axiom MSG: notice that $\mathrm{dom}(\Delta) \subseteq \mathrm{dom}(\Gamma \restriction \mathrm{n}(a!l[\tilde{v}]))$ and that the axiom makes $\{a, \tilde{v}\} \subseteq \mathrm{dom}(\Delta)$. We show that $\Delta \stackrel{\mathrm{def}}{=} \{a{:}(\mathbf{0}, \upsilon.l(\widetilde{\mathbf{0}})), \tilde{v}{:}(\mathbf{0}, \widetilde{\mathbf{0}})\}$ is a minimal typing of P. Clearly, $\upsilon.l(\widetilde{\mathbf{0}})$ respects the side condition of the axiom, hence $\Delta \vdash P$. To see that it is minimal, observe that, by Lemma 8.3.3, there is a normal form β such that $\Gamma(a) \approx \beta$, and by hypothesis $\Gamma(a)_{\mathsf{o}} \Longrightarrow\xrightarrow{l(\widetilde{\mathbf{0}})}$, hence $\beta \Longrightarrow\xrightarrow{l(\widetilde{\beta})}$ with $\widetilde{\beta} \approx \widetilde{\mathbf{0}}$. We do now a case analysis on the form of β:
      i. case β is a labelled sum or a parallel composition, then $\beta \leq l(\widetilde{\beta}) \leq \upsilon.l(\widetilde{\beta})$;
      ii. case β is a blocked sum, then $\beta \leq \approx \upsilon.l(\widetilde{\beta})$.

   Since $\approx$ preserves $\leq$ (Proposition 9.3.2), the result follows.
   (b) Axiom VAR: the minimal type of each v in $\vec{v}$ should be t; notice that $t \approx \mathbf{0}$ and $\mathbf{0}$ is the top of the types lattice, hence, $\Delta(\vec{v}) = B(X) = \vec{t}$ and $\mathrm{dom}(\Delta) = \{\vec{v}\}$.

   The interesting cases of the induction step are the rules OBJ, REC, and PAR. We construct the minimal typings for each case below; one easily verifies that they are under the conditions of the definition.

   Induction step:
   (a) Rule OBJ: notice that $\mathrm{dom}(\Delta) \subseteq \mathrm{dom}(\Gamma \restriction \mathrm{n}(P)) \setminus \bigcup_{i \in I}\{\tilde{x}_i\}$.
   By hypothesis, $\forall_{i \in I}\, B, \Gamma \cdot \tilde{x}_i{:}(\mathbf{0}, \widetilde{\beta_i}) \vdash P_i$, hence by induction hypothesis, there are minimal typings $B, \Delta_i \cdot \tilde{x}_i{:}(\mathbf{0}, \widetilde{\beta'_i}) \vdash P_i$, for each $i \in I$. The minimal typing Δ is such that $\Delta(a)_{\mathsf{i}} = \sum_{i \in I} \upsilon.l_i(\widetilde{\beta'_i}).\Delta_i(a)_{\mathsf{i}}$, $\Delta(a)_{\mathsf{o}} = \sum_{i \in I} \upsilon.\Delta_i(a)_{\mathsf{o}}$, and furthermore $\forall_{i \in I} \forall b \in \mathrm{dom}(\Delta_i)$ with $b \neq a$, $\Delta(b)_{\mathsf{i}} = \sum_{i \in I} \upsilon.\Delta_i(b)_{\mathsf{i}}$ and $\Delta(b)_{\mathsf{o}} = \sum_{i \in I} \upsilon.\Delta_i(b)_{\mathsf{o}}$. Obviously, Δ respects the side conditions of this rule, thus $B, \Delta \vdash a?\{l_i(\tilde{x}_i) = P_i \mid i \in I\}$. To check that Δ is minimal recall that all Δ_i are minimal; clearly $\Gamma(a) \leq \Delta(a)$ and $\Gamma(b) \leq \Delta(b)$, and both are the simplest types respecting the conditions of this rule.

(b) Rule REC: since we set $\mathrm{fn}(a?M) \subseteq \{\vec{x}\}$ (page 19), $\{\vec{v}\} \subseteq \mathrm{dom}(\Delta)$ (the possibly remaining names are bound in $a?M$). By induction hypothesis, there is a minimal typing $\Delta' \cdot X{:}\vec{\gamma} \cdot \vec{x}{:}\vec{\gamma} \vdash a?M$ where t occurs in each γ in $\vec{\gamma}$ (cf. done in axiom VAR). Thus, $\Delta(\vec{v}) = \mu t.\vec{\gamma}$. Like in the previous case, one easily sees that Δ respects the side conditions of this rule and it is minimal.

(c) Rule PAR: simply make the pairwise parallel composition of the minimal typings of the components. The result is clearly a minimal typing for the parallel composition.

2. (Uniqueness) Suppose there is a typing Γ', different from Γ which is also a minimal typing of P. Then, by definition, $\Gamma \leq \approx\Gamma'$, but since, by hypothesis, Γ minimal, $\Gamma' \leq \approx\Gamma$ holds as well. Therefore, we conclude that $\Gamma \approx \Gamma'$.

□

PROPOSITION 14.3.8 (RECURSIVE OUTPUT TYPES).
Take a minimal Γ for P. If, for some name a, the type $\Gamma(a)_o$ has a subterm of the form $\mu t.\beta$ where t is free in β, then β is a blocked sum.

PROOF. Recall that we consider that process variables only occur bound in processes. The proof requires a case analysis of the type system's rules, assuming Γ minimal for P, by Proposition 14.3.7.1. Notice that the side conditions of rule REC require that the types of the free output names of $(\textbf{rec}\, X(\vec{x}).a?M)[\vec{v}]$ have $\mu t. \sum_{i\in I} \upsilon.\beta_i$ as supertype. □

14.3.3 PRESERVATION OF TYPABILITY

We proceed now with the proof of the soundness of the type system. We establish a correspondence between process transitions and type transitions, and characterise persistent bad redexes in terms of the agreement and the compatibility relations.

LEMMA 14.3.9 (OPERATIONAL CORRESPONDENCE).
Let Γ be a minimal typing of P.

1. $\Gamma(a)_o \Longrightarrow\xrightarrow{l(\widetilde{\beta})}$ *with* $\Gamma(\tilde{v})_o \approx \widetilde{\beta}$, *if* $P \equiv \nu\tilde{x}\, a!l[\tilde{v}] \mid P'$;

2. $\Gamma(a)_i \Longrightarrow\xrightarrow{l_i(\widetilde{\beta}_i)}$, *with* $\Delta(\tilde{x}_i)_o \approx \widetilde{\beta}_i$ *where* $X{:}\vec{\gamma}, \Delta \vdash P_i$ *for any* $i \in I$, *if*
either $P \equiv \nu\tilde{u}\, a?\{l_i(\tilde{x}_i) = P_i \mid i \in I\} \mid P'$
or $P \equiv \nu\tilde{u}\, (\textbf{rec}\, X(x\vec{x}).x?\{l_i(\tilde{x}_i) = P_i \mid i \in I\})[a\vec{v}] \mid P'$;

3. $\Gamma(a)_i \Longrightarrow\xrightarrow{l_i(\widetilde{\beta}_i)}$, *with* $\Delta(\tilde{x}_i)_o \approx \widetilde{\beta}_i$ *where* $X{:}\vec{\gamma}, \Delta \vdash P_i$ *for any* $i \in I$,
and $\Gamma(a)_o \Longrightarrow\xrightarrow{l(\widetilde{\beta})}$, *with* $\Gamma(\tilde{v})_o \approx \widetilde{\beta}$, *if*
either $P \equiv \nu\tilde{u}\, a?\{l_i(\tilde{x}_i) = P_i \mid i \in I\} \mid a!l[\tilde{v}] \mid P'$
or $P \equiv \nu\tilde{u}\, (\textbf{rec}\, X(x\vec{x}).x?\{l_i(\tilde{x}_i) = P_i \mid i \in I\})[a\vec{v}] \mid a!l[\tilde{v}] \mid P'$.

PROOF.

1. By typability of subterms, observing that the side condition of the axiom MSG ensures the result (notice that Γ does not need to be a minimal typing of P).
2. Similar to the previous clause, but taking into account that Γ is a minimal typing of P.
3. Follows from the previous clauses.

$\square$

PROPOSITION 14.3.10 (PRESERVATION OF TYPABILITY).
Let Γ be a minimal typing of P.

1. $\Gamma(a)_o \Longrightarrow\xrightarrow{l(\widetilde{\beta})} \approx\Delta(a)_o$ *with* Δ *being a minimal typing of* Q, $\Gamma \setminus a \approx \Delta \setminus a$, *and* $\Gamma(a)_i \approx \Delta(a)_i$, *if* $P \xrightarrow{\nu\tilde{x}\, a!l[\tilde{v}]} Q$;
2. $\Gamma(a)_i \Longrightarrow\xrightarrow{l(\widetilde{\beta})} \approx\Delta(a)_i$, *with* $\tilde{v}$ *and* $\widetilde{\beta}$ *having the same length,* Δ *being a minimal typing of* Q, $\Delta(\tilde{v})_o \approx \Gamma(\tilde{v})_o \parallel \widetilde{\beta}$, $\Delta(\tilde{v})_i \approx \Gamma(\tilde{v})_i$, *and* $\Gamma \setminus \tilde{v} \approx \Delta \setminus \tilde{v}$, *if* $P \xrightarrow{a?l[\tilde{v}]} Q$;
3. $\Gamma(a)_o \Longrightarrow\xrightarrow{l(\widetilde{\beta})} \approx\Delta(a)_o$ *and* $\Gamma(a)_i \Longrightarrow\xrightarrow{l(\widetilde{\beta})} \approx\Delta(a)_i$, *with* $\tilde{v}$ *and* $\widetilde{\beta}$ *having the same length,* Δ *being a minimal typing of* Q, *and* $\Gamma \setminus a \approx \Delta \setminus a$, *if* $P \xrightarrow{\tau} Q$.

PROOF. Straightforward, by a case analysis on $P \xrightarrow{m} Q$, using Lemma 3.3.2, the appropriate rules of Tables 3.1 and 14.1, and Lemma 14.3.9. $\square$

PROPOSITION 14.3.11 (BAD REDEXES).
Let Γ be a minimal typing of P. For some $a \in \mathrm{dom}(\Gamma \upharpoonright \mathrm{n}(P))$,

1. $\Gamma(a)_i \not\bowtie^P_\Gamma \Gamma(a)_o$, *if* $P \Longrightarrow Q$ *and* Q *has a persistent bad redex;*
2. $\Gamma(a)_i \not\lhd^P_\Gamma \Gamma(a)_o$, *if, for some non-empty input sequence* $\widetilde{m}$ *and process* Q, $P \stackrel{\widetilde{m}}{\Longrightarrow} Q$ *and* Q *has a persistent bad redex not generated by* $\widetilde{m}$.

PROOF. Notice that the negation of Definition 13.2.1 makes $\Gamma(a)_{\mathsf{i}}, \Gamma(a)_{\mathsf{o}} \not\approx \mathbf{0}$, and ensures that some transition of $\Gamma(a)_{\mathsf{i}}$ is not matched by $\Gamma(a)_{\mathsf{o}}$.

1. By induction on the length of $\Longrightarrow$.

   Case $P \equiv Q$ then, by Definition 5.1.2 and by Lemma 14.3.9.3, the result follows.

   Case $P \stackrel{v}{\Longrightarrow} Q$, by Proposition 14.3.10.3, there is a minimal Δ of Q, such that, for some $\widetilde{\pi}$, $\Gamma(a) \stackrel{\widetilde{\pi}}{\Longrightarrow} \approx\Delta(a)$; by induction hypothesis $\Delta(a)_{\mathsf{i}} \not\bowtie^Q_\Delta \Delta(a)_{\mathsf{o}}$, and the result follows.

2. Recall that $P \overset{\widetilde{m}}{\Longrightarrow} Q$ denotes $P \Longrightarrow P_1 \xrightarrow{m_1} P_1' \Longrightarrow \cdots \Longrightarrow P_n \xrightarrow{m_n} P_n' \Longrightarrow Q\,,\ n \geq 1$. We proceed by induction on the sum of the lengths of $\Longrightarrow$ and of $\widetilde{m}$.

   Base case: $P \equiv P_1,\ n = 1$, and $P_1' \equiv Q$; by Proposition 14.3.10.2, by Definition 5.1.2, and by Lemma 14.3.9.3, for some $c \in \mathrm{dom}(\Delta)$, we have $\Delta(c)_{\mathsf{i}} \not\bowtie^{Q}_{\Delta} \Delta(c)_{\mathsf{o}}$, and the result follows.

   The induction step is similar to that of the previous case.

$\square$

We are finally in a position to ensure the preservation of the typability of a process by the transition relation.

Theorem 14.3.12 (Subject reduction).
Consider that Γ is a well-typing of process P.

1. *If $P \xrightarrow{\tau} Q$, then Q is well-typed.*
2. *If $P \xrightarrow{\nu\tilde{x}\ a!l[\tilde{v}]} Q$, then Q is well-typed.*
3. *If $P \xrightarrow{a?l[\tilde{v}]} Q$, then there is a minimal typing Δ of Q such that $(\Delta(\tilde{v})_{\mathsf{i}} \prec^{Q}_{\Delta} \Delta(\tilde{v})_{\mathsf{o}} \parallel \widetilde{\beta}$ and if $a \notin \{\tilde{v}\}$ then $\Delta(a)_{\mathsf{i}} \prec^{Q}_{\Delta} \Delta(a)_{\mathsf{o}})$ implies Q is well-typed.*

Proof. By a case analysis on $P \xrightarrow{m} Q$, using Proposition 14.3.10 and Lemma 13.2.4. $\square$

This last result guarantees that the type system respects the notion of error, accepting only error-free processes.

Corollary 14.3.13 (Soundness of the type system).
Well-typed processes are error-free.

Proof. Assume that for some Q and $\widetilde{m}$, $P \overset{\widetilde{m}}{\Longrightarrow} Q$ where Q has a bad $al\tilde{v}$-redex not generated by $\widetilde{m}$. We show that there is an R and an $\widetilde{n}$ such that $Q \overset{\widetilde{n}}{\Longrightarrow} R$ where R does not have a bad $al\tilde{v}$-redex. The proof is by induction on the sum of the lengths of $\widetilde{m}$ and $\widetilde{n}$.

Base case: $\widetilde{m}$ and $\widetilde{n}$ both have length zero;

so $P \Longrightarrow Q$, and by successive applications of the Theorem 14.3.12.1, we conclude that Q is well-typed. Thus, by Proposition 14.3.10.3, take Γ as a well typing of Q; since we assume that Q has a bad $al\tilde{v}$-redex, it holds that

$$\Gamma(a)_{\mathsf{i}}, \Gamma(a)_{\mathsf{o}} \neq \mathbf{0}, \text{ and } l(\widetilde{\beta}) \in \mathrm{int}(\Gamma(a)_{\mathsf{o}}), l(\widetilde{\beta}) \notin \mathrm{int}(\Gamma(a)_{\mathsf{i}}) \neq \mathbf{0} \text{ where } \tilde{v}{:}(\_, \widetilde{\beta}). \quad (14.1)$$

To proceed, we consider separately the cases of a being bound or free.

1. Case $a \in \mathrm{bn}(Q)$.
   Since Q is well-typed, $\Gamma(a)_{\mathsf{i}} \bowtie_{\Gamma}^{Q} \Gamma(a)_{\mathsf{o}}$, and observe that, if $Q \not\xrightarrow{m}$ for any m, the bad redex is persistent, and we reach an absurd using Proposition 14.3.11. So there is an $l'(\widetilde{\beta'})$ with $l' \neq l$ or $|\widetilde{\beta'}| \neq |\tilde{v}|$, such that conditions 1(b)i and 1(b)ii of Definition 13.2.1 hold (notice that condition 1(b)iii holds trivially, due to the assertion 14.1 above). Thus, as $Q \xrightarrow{m}$, the contrapose of Proposition 14.3.11.1 ensures that Q has no persistent $al\tilde{v}$-redex.
2. Case $a \in \mathrm{fn}(Q)$.
   Since Q is well-typed, $\Gamma(a)_{\mathsf{i}} \prec_{\Gamma}^{Q} \Gamma(a)_{\mathsf{o}}$, so there is a $\widetilde{\pi}$ and an $l'(\widetilde{\beta'})$, with $l' \neq l$ or $|\widetilde{\beta'}| \neq |\tilde{v}|$, such that conditions 1(b)i and 1(b)ii of Definition 13.2.1 hold (again, condition 1(b)iii holds trivially due to the assertion 14.1 above). Proceeding as above we conclude that Q has no persistent $al\tilde{v}$-redex.

Induction step: We still have two cases to consider.

1. Case $\widetilde{m}$ has length zero, but $\widetilde{n}$ has length greater than zero.
   Let $P \Longrightarrow Q \xrightarrow{a?l[\tilde{v}]} S$ where Q is well-typed by Theorem 14.3.12.1. Since $a \in \mathrm{fn}(Q)$, $\Gamma(a)_{\mathsf{i}} \prec_{\Gamma}^{Q} \Gamma(a)_{\mathsf{o}}$, for some well-typing Γ of Q, and by Proposition 14.3.10. 2, $\Gamma(a)_{\mathsf{i}} \Longrightarrow\xrightarrow{l(\widetilde{\beta})} \approx \Delta(a)_{\mathsf{i}}$ where Δ is a minimal typing of S and $\Delta(\tilde{v})_{\mathsf{o}} \approx \Gamma(\tilde{v})_{\mathsf{o}} \parallel \widetilde{\beta}$. Since Γ is a well typing of Q, also $\Delta(a)_{\mathsf{i}} \prec_{\Delta}^{S} \Delta(a)_{\mathsf{o}}$. If $\Delta(\tilde{v})_{\mathsf{i}} \prec_{\Delta}^{S} \Delta(\tilde{v})_{\mathsf{o}}$, by Theorem 14.3.12.3, S is well-typed, and the result follows using the induction hypothesis; otherwise, as Γ is a well typing of Q, $\Gamma(\tilde{v})_{\mathsf{i}} \prec_{\Gamma}^{Q} \Gamma(\tilde{v})_{\mathsf{o}}$, but $\Gamma(\tilde{v})_{\mathsf{i}} \prec_{\Gamma}^{Q} \Gamma(\tilde{v})_{\mathsf{o}} \parallel \widetilde{\beta}$. Therefore, again using the contrapose of Proposition 14.3.11.2, the input action $a?l[\tilde{v}]$ generates the bad redex from Q, and then we are not under the conditions of Definition 5.1.7 in page 41, so the proof is complete.
2. Case $\widetilde{m}$ and $\widetilde{n}$ both have length greater than zero. The proof is similar to that of the previous case.

$\square$

Notice that the converse of this result is not true. There are several examples of "good" processes (with respect to the notion of error) which are not typable. The agreement and compatibility relations do not have causality information, thus there are deadlocks which are not errors and are detected (a nice side-effect — cf. Example 14.1.3.3(b)), but there are interesting processes that are rejected (cf. Example 14.1.3.3(c)).

The following result, despite not being surprising in our setting, is a relevant property for a type system.

THEOREM 14.3.14 (UNIQUENESS OF TYPINGS).
Consider that Γ is a well-typing of process P. If $\Gamma \approx \Delta$, then Δ is a well-typing of P.

PROOF. By induction on the deduction of $\Gamma \vdash P$, using rule $\approx$ and Proposition 13.2.3. $\square$

Part V

Conclusions

Chapter 15

Related work

In sequential settings, type theory is based on the λ-calculus. The work on types in λ is a strong inspiration to the work on types in process calculi. The motives to investigate types in this context are threefold:

1. to guarantee programme safety through Subject Reduction, or even stronger properties like deadlock-freedom or the prevention of information leakage;
2. to provide insight for efficient implementations;
3. to define typed equivalences that ensure stronger behavioural properties on processes than on untyped settings.

Herein we make an overview on types for name-passing process calculi, including works on non-uniform types. The work reported should be straightforwardly adapted to TyCO, as encoding π into it is a simple task. However, TyCO is not π, and making comparisons with π, or adapting our work to π, is not a simple task. The definition of error used in this thesis is difficult to express in π, if indeed possible. The problem of encoding π-calculus in TyCO is under investigation [NR00].

15.1 Work on types for name-passing calculi

The original presentation of the π-calculus is monadic, i.e. only one name at a time may circulate in a channel. Polyadic π is an extension where finite tuples of names circulate in channels [Mil93]. Naturally, a run-time error may occur, if a tuple of the wrong arity is sent to a given name. Take process $a(x\,y).P \mid \overline{a}[b]$. The name a is expecting two names, but receives only one, this situation leading to a run-time error. Polyadic π is implicitly typed: it is possible to statically prevent such a situation, without the need of including type annotations in the processes.

Type checking is a mechanism that statically detects possible run-time errors. In the context of name-passing calculi, a type system imposes a discipline on how should names carry other names, in a fixed way. Richer notions of types provide valuable information for other uses, like code optimisation, or to ensure properties like deadlock-freedom.

A general study on a type theory for name-passing calculi is Honda's theory of types for the π-calculus as a certain partial algebra [Hon96], a framework to investigate type systems and process composition in typed environments. König proposes an hypergraph-based process calculus [Kön99a, Kön99b], akin to asynchronous π, with a generic type system that can be instantiated to guarantee some of the specific properties of type systems described ahead. The systems enjoys the Subject Reduction property, the existence of principal types, and of a type inference algorithm.

An abridged summary of increasingly more complex type systems for π follows. It is based on Honda's selected bibliography on mobile processes.[1] Two aspects are left aside: type systems for distributed calculi and for secrecy.

Sorting. On his tutorial on Polyadic π, Milner proposes *sorting*, a system to detect arity mismatch. From a basic collection of subject sorts and from a sorting — a total function of sorts into sequences of sorts — Milner defines a type system that assigns sorts to the free names of a process. The system enjoys a Subject Reduction property, and supports recursion.

Gay developed a type inference algorithm for sorting [Gay93]. Independently, Vasconcelos and Honda gave a typing system for sorting, together with an efficient typing algorithm [VH93].

Polymorphism is also considered in sorting. Turner studies impredicative polymorphism, encoding the System F in polymorphic pi-calculus [Tur96]. Vasconcelos proposes a predicative polymorphism with ML-like type inference [Vas94b].

Input/Output modes. An important improvement on sorting is to include information on whether a process uses a name for input or output [PS96]. Type systems with input/output modes distinguish the ability of reading from a channel, from that of writing into a channel. A subtyping relation arises naturally, as "a process can always behave more stringently than specified by the type".

The system prevents interference and improves dataflow information. It is used on proving the validity of β-reduction for the more efficient of Milner's encoding of the call-by-value λ-calculus, a property that does not hold in "sorted" π.

[1] URL: http://www.cs.auc.dk/mobility/bib/honda.html

Linearity. Another extension to sorting is to add information on how many times is a channel used. Types are annotated with a *multiplicity* that is either 1 — used only once — or w — used any number of times. In particular, a linear usage of a name guarantees a reply, reduces non-determinism — reductions on linear channels follow a partial confluence result, simplifies reasoning about congruences by reducing contexts, and leads to heap usage optimisations.

Linear types were proposed by Kobayashi, Pierce, and Turner [KPT96]. Sangiorgi refines the notion defining *uniform receptiveness* [San99a]: "a name x is uniformly receptive in a process P if: (1) at any time P is ready to accept an input at x, at least as long as there are processes that could send messages at x; and (2) the input offer at x is functional". He defines a theory and proof techniques for uniform receptiveness, and uses it on proving the correctness on some program transformations [San99b].

Deadlock detection. In the context of the π-calculus, there is some work on deadlock detection, that is, a message sent but never received, or a receiver waiting for a message that never arrives. Kobayashi *et al.* guarantee partial deadlock freedom and partial confluence of programs [Kob97, SK98], via a type system that generalises the input/output modes and linearity. Instead of multiplicities, types are annotated with a *usage* that is a term of a very simple process algebra build from 0, input prefixing, output prefixing, parallel composition, and 'bang' (any number of times). A usage specifies in what order and how should a channel be used. Moreover, input and output prefixes are further annotated with an attribute that is either a *capability* — communication on the channel *may* be provided — or an *obligation* — communication on the channel *must* be provided. A channel is reliable if its capabilities and obligations are consistent. Finally, time tags and a tag relation establish an ordering of usages, which together with usage reliability ensures deadlock-freedom.

Yoshida defines graph types for monadic π-processes, directed graphs where the nodes denote basic actions — usage of a channel as input or output and type of the value passed — and the edges denote the ordering between the actions [Yos96]. The graphs characterise the dynamics of the processes, capturing some deadlocks, since processes can only be composed when they use common channels in a complementary way.

Behavioural types. All the type systems previously referred but Yoshida's one share a property: types are static and rigid, and do not allow that names may change their behaviour. Honda *et al.* [HVK98, THK94] propose *session types* that specify sequences of message types. The former are input/output types; the latter embodies the former as basic types, and are constructed with a sequencing operator, and with a choice operator, giving types a branching structure. Therefore, session types are adequate to describe client-server protocols.

Gay and Hole [GH99] extended session types with subtyping, obtaining a system that supports the incremental development of protocols by refinement.

15.2 Non-uniform types

There are several works on non-uniform types for mobile processes. In all of them the notion of run-time error is the static 'message-not understood'.

1. Boudol proposes a dynamic type system for the Blue Calculus, a variant of the π-calculus directly incorporating the λ-calculus [Bou97]. The types are functional, in the style of Curry simple types, and incorporate Hennessy-Milner logic with recursion — modalities interpreted as resources of names. Processes inhabit the types, and this approach captures some causality in the usage of names in a process, ensuring that messages to a name will meet a corresponding offer. Well-typed processes behave correctly, this being preserved under reduction.
2. Colaço *et al.* [Col97, CPS97, CPDS99] propose a calculus of actors based on an extended TyCO, and a type system which aims at the detection of "orphan messages", i.e. messages that may never be accepted in some actor's execution path, either because the requested service is not in the actor's interface, or due to dynamic changes in a actor's interface. Types are interface-like, with multiplicities, therefore without dynamic information, and the type system requires complex operations on a lattice of types. The type inference algorithm is a set-constraints algorithm, a well-known technique widely used in functional languages [AW93].
3. Najm and Nimour [NN97, NNS99a, NNS99b] propose several versions of a calculus of objects that features dynamically changing interfaces and distinguishes between private and public objects' interfaces. For each version of the calculus they develop a typing system handling dynamic method offers in private interfaces, and guaranteeing a liveness property: all pending requests are treated. Types are sets of deterministic guarded parametric equations, equipped with a transition relation, and representing infinite state systems. A type inference algorithm is built on an equivalence relation, a compatibility relation, and a subtyping relation on types, based on the simulation and on the bisimulation relations (strong versions — decidable relations).
4. Puntigam [Pun97, Pun99] defines a calculus of active objects and process types. A type system ensures that all messages sent to an object are received and answered. Objects are syntactically constrained to a unique identity and messages are received in the order they were sent, as every object is associated with a FIFO queue. The expressiveness of types is that of a non-regular language. Subtyping is supported.

Chapter 16

Concluding remarks

In this thesis we aim at studying non-uniform objects in TyCO, and at obtaining a type system to ensure the safety of programmes with non-uniform objects. This last chapter presents a short summary on the work done and on the work that is left to be done.

16.1 Achievements

From the beginning it was clear that a dynamic notion of processes with communication errors makes sense. It leads to a more natural style of programming than, for example, busy waiting, and moreover, programmes are "behaviour-oriented". The motto is *errors are persistent bad redexes.* Inspired by Nierstrasz's regular types, we develop a notion of type that is a concise specification of the behaviour of a non-uniform object, as a type describes the sequences of messages that an object accepts and its blocking states. The algebra of types expresses the dynamics of non-uniform objects. The transitional relations agreement and compatibility capture the two situations provoking persistent bad redexes. Building a sound type system is almost straightforward, but the proofs are a bit hard.

Therefore, apart from the exercise of modelling non-uniform objects and compare them to their uniform versions, understanding the pros and cons of non-uniformity, the contributions of this thesis are:

1. the novel notion of process with a communication error, in short, with a persistent bad redexes;
2. the algebra of behavioural types, ABT, and its equivalence notion, for which we define an axiomatic system, complete at least for image-finite processes;
3. the type system, sound with respect to the notion of error, which is based on the transitional relations agreement and compatibility that hold only when processes are error-free.

16.2 Future work

Several open issues on TyCO are worthwhile to investigate, although they are not related to this thesis. One trend is on behavioural equivalences and their axiomatic characterisations. Another is the study of encodings of TyCO into the π-calculus. It would also be interesting to understand the expressive power of each of the operators providing potentially unbound behaviour, maybe through mutual encodings, as well as to study the encoding of polyadic TyCO into monadic, using ABT.

With respect to non-uniform objects in process calculi, I may say that now I am starting to understand the problem. The solutions presented herein are preliminary and not totally satisfactory. Many aspects remain for further investigation.

Abstract machine. The present TyCO compiler is not ready to deal with non-uniform objects. We started the study of an abstract machine for non-uniform TyCO, having defined the rules. The correctness proofs remain to be done, and an implementation crucially depends on an efficient data-structure to code the run-queue.

Completeness and decidability of ABT. We still do not have a final answer to the problem of the completeness of the axiomatic system of *lsb*. Furthermore, it is important to find out whether *lsb* is decidable, relating the work to that on the decidability of weak notions of bisimulation.

Modal Logics. Another aspect that deserves attention is the modal characterisation of *lsb*, and the use of a modal logic to express and verify safety and liveness properties.

Algorithm for type inference. The existence of an algorithm to infer the minimal typing of a process is not clear. Albeit the rules are simple, the algorithm depends on decision procedures for *lsb*, for subtyping, and for the agreement and compatibility relations. Without these results, the type system is useless. If they turn out to be impossible to obtain, a reformulation of *lsb* is mandatory.

Deadlock detection. It is equally interesting to study how to enrich the type system to reject less processes, and to detect deadlocks. It is also interesting to study refinements of the system to encompass linearity and receptiveness, since the structure of types posses enough information to treat these aspects.

Part VI

Appendix

Appendix A

On completeness for concurrent finite types

We present here a discussion on a possible alternative proof of Theorem 8.3.5 that does not require the inference rule 8.1, and uses only induction.

Conjecture A.1 (Op-congruence of parallel).
If $\alpha \parallel \beta$ and $\alpha' \parallel \beta'$ are normal forms, and $\alpha \approx \alpha'$, then $\alpha \parallel \beta \approx \alpha \parallel \beta'$ implies $\beta \approx \beta'$.

Discussion. Since $\alpha \parallel \beta$ and $\alpha \parallel \beta'$ are both normal forms and $\alpha \approx \alpha'$, then, either α is a blocked sum and β and β' are both labelled sums, or vice-versa; moreover, we have $\alpha, \beta, \beta' \not\approx \mathbf{0}$. We proceed now by induction on $\mathrm{depth}(\alpha) + \mathrm{depth}(\beta) + \mathrm{depth}(\beta')$. The base cases are when the sum of depths is 4 or 5:

1. if α is a blocked sum with $\mathrm{depth}(\alpha) = 2$ then β and β' are labelled sums of depth 1; the result is immediate;
2. if α is a labelled sums with $\mathrm{depth}(\alpha) = 1$ then β and β' are blocked sums of depth 2; the result follows by the definition of $\approx$.

For the induction step we do a case analysis on the structure of the types.

1. Case α is blocked, with all its derivatives not being parallel compositions, by Lemma 8.3.4.

   As it is a normal form, it is saturated, hence if $\alpha \xrightarrow{v} \alpha_i$ then $\exists \alpha_j\, \alpha \xrightarrow{v} \alpha_j$ and $\alpha_i \parallel \beta \approx \alpha_j \parallel \beta'$.

   If $\alpha_i \approx \mathbf{0}$ then, by the Proposition 8.2.3, $\alpha_j \approx \mathbf{0}$ and the result follows immediately.

   Otherwise, if $\alpha_i \approx \alpha_j$ then the result follows by induction.

   The remaining case ($\alpha_i \not\approx \alpha_j$) should be impossible, but we still did not prove it.
2. Case α is a labelled sum, we proceed similarly to the previous case. $\square$

The proof of the following conjecture depends on the previous conjecture holding.

Conjecture A.2.
Consider types $\alpha \equiv \alpha_1 \parallel \alpha_2$ and $\beta \equiv \beta_1 \parallel \beta_2$ normal forms, with α_1 and β_1 labelled sums. Then, $\alpha \approx \beta$ if, and only if, $\alpha_1 \approx \beta_1$ and $\alpha_2 \approx \beta_2$.

Discussion. The 'if' direction is a consequence of Proposition 7.3.4. For the 'only-if' direction notice first that if $\alpha_2 \approx \mathbf{0}$ (or β_2), the result follows by the Lemma 8.2.3. Notice that if we consider $\alpha_1 \approx \beta_1$ but $\alpha_2 \not\approx \beta_2$ or vice-versa then by the contrapositive of the previous lemma we conclude an absurd; moreover, clearly $\alpha_1 \not\approx \beta_2$ and $\alpha_2 \not\approx \beta_1$. From here the proof follows by induction on $\mathrm{depth}(\alpha_1) + \mathrm{depth}(\beta_1)$. The base is $\mathrm{depth}(\alpha_1) = \mathrm{depth}(\beta_1) = 0$; we immediately obtain $\alpha_1 \approx \beta_1 \approx \mathbf{0}$ and thus, $\alpha_2 \approx \beta_2$. For the induction step assume $\mathrm{depth}(\alpha_1) + \mathrm{depth}(\beta_1) > 0$. Then $\alpha_1 \parallel \alpha_2$ and $\beta_1 \parallel \beta_2$ are strictly unblocked, and in fact the sum of the depths is at least 2 — if it is exactly 2 the result follows immediately, therefore we consider it bigger than 2.

So, as $\alpha \approx \beta$, there are l-derivatives $\alpha_i, \beta_k \not\approx \mathbf{0}$ of α_1, β_1 such that $\alpha_i \parallel \alpha_2 \approx \beta_k \parallel \beta_2$ holds; case α_i and β_k are bisimilar labelled sums, by induction we obtain $\alpha_2 \approx \beta_2$, and thus, by Conjecture A.1, also $\alpha_1 \approx \beta_1$; case α_i and β_k are bisimilar blocked sums, again by Conjecture A.1, $\alpha_2 \approx \beta_2$, and $\alpha_1 \approx \beta_1$.

It is also the case that there are υ-derivatives α_j, β_l of α_2, β_2, such that $\alpha_1 \parallel \alpha_j \approx \beta_1 \parallel \beta_l$ holds, with $\alpha_1 \not\approx \beta_1$ and $\alpha_2 \not\approx \beta_2$; the proof follows similarly to the previous case. □

Notice that the previous conjecture is false in general, as the following examples show:

1. $(\upsilon.l \parallel \upsilon.m) \parallel \upsilon.n \approx \upsilon.l \parallel (\upsilon.m \parallel \upsilon.n)$,
2. $(l \parallel m) \parallel n \approx l \parallel (m \parallel n)$, and
3. $(l \parallel \upsilon.m) \parallel \upsilon.n \approx l \parallel (\upsilon.m \parallel \upsilon.n)$.

The following discussion is only a proof of Theorem 8.3.5, if Conjecture A.1 holds (thus, Conjecture A.2 also holds). We strongly believe that indeed it is the case.

Discussion. The proof of the completeness result is by induction on the sum of the depths of the types α and β (assumed to be normal forms, by Lemma 8.3.3). Taking into account the proof of Theorem 8.3.5, the only case to consider is when $\alpha \equiv \sum_{i\in I} l_i(\widetilde{\alpha}_i).\alpha_i \parallel \sum_{j\in J} \upsilon.\alpha_j$ and $\beta \equiv \sum_{k\in K} m_k(\widetilde{\beta}_k).\beta_k \parallel \upsilon.\beta_l$ where $I, K \neq \emptyset$.

1. if $J = \emptyset$, $L \neq \emptyset$ then the result follows by Lemma 8.2.3;
2. if $J, L \neq \emptyset$ then the result follows by Conjecture A.2. □

Bibliography

[Abr90] Samson Abramsky. The lazy lambda-calculus. In *Research topics in Functional Programming*, pages 65–117. Addison-Wesley, 1990.

[AC96] Martin Abadi and Luca Cardelli. *A Theory Of Objects*. Springer-Verlag, 1996.

[ACS98] Roberto Amadio, Ilaria Castellani, and Davide Sangiorgi. On bisimulations for the asynchronous π-calculus. *Theoretical Computer Science*, 195(2):291–324, 1998. An extended abstract appeared in *Proceedings of CONCUR'96*, volume LNCS 1119, 147–162.

[Agh86] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. M. I. T. Press, 1986.

[Ama97] Roberto Amadio. An asynchronous model of locality, failure, and process mobility. In David Garlan and Daniel Le Metayer, editors, *Proceedings of COORDINATION'97*, volume 1282 of *Lecture Notes in Computer Science*, pages 374–391. Springer-Verlag, 1997. Extended version as Rapport de Recherche RR-3109, INRIA, Sophia-Antipolis.

[Ame89] Pierre America. Issues in the design of a parallel object-oriented language. *Formal Aspects of Computing*, 1(4):366–411, 1989.

[AW93] Alexander Aiken and Edward L. Wimmers. Type inclusion constraints and type inference. In *Functional Programming & Computer Architecture*, pages 31–41. ACM Press, 1993.

[Bar84] Henk Barendregt. *The Lambda Calculus - Its Syntax and Semantics*. North-Holland, 1981 (1st ed.) revised 84.

[BB92] Gérard Berry and Gérard Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96:217–248, 1992.

[BBK87] Jos C. M. Baeten, Jan A. Bergstra, and Jan W. Klop. On the consistency of Koomen's fair abstraction rule. *Theoretical Computer Science*, 51(1–2):129–176, 1987.

[BBSDS97] Howard Bowman, Charles Briscoe-Smith, John Derrick, and Ben Strulo. On behavioural subtyping in LOTOS. In Bowman and Derrick [BD97].

[BD97] Howard Bowman and John Derrick, editors. *2nd IFIP Conference on Formal Methods for Open Object-based Distributed Systems, Canterbury, U. K.* Chapman & Hall, 1997.

[BE97] Olaf Burkart and Javier Esparza. More infinite results. *Bulletin of the European Association for Theoretical Computer Science (EATCS)*, 62:138–159, 1997.

[Bes93] Eike Best, editor. *4th International Conference on Concurrency Theory (CONCUR), Hildesheim, Germany*, volume 1243 of *Lecture Notes in Computer Science*. Springer-Verlag, 1993.

[BK85] Jan A. Bergstra and Jan W. Klop. Algebra of communicating processes with abstraction. *Theoretical Computer Science*, 37(1):77–121, 1985.

[Bor98] Michele Boreale. On the expressiveness of internal mobility in name-passing calculi. *Theoretical Computer Science*, 195(2):205–226, 1998. An extended abstract appeared in *Proceedings of CONCUR'96*, volume LNCS 1119, pages 163–178.

[Bou92] Gérard Boudol. Asynchrony and the π-calculus (note). Rapport de Recherche 1702, INRIA Sophia-Antipolis, 1992.

[Bou97] Gérard Boudol. Typing the use of resources in a concurrent calculus. In R. K. Shyamasundar and Kazunori Ueda, editors, *Proceedings of ASIAN'97*, volume 1345 of *Lecture Notes in Computer Science*, pages 239–253. Springer-Verlag, 1997.

[Bou98] Gérard Boudol. The π-calculus in direct style. *Higher-Order and Symbolic Computation*, 11:177–208, 1998. An extended abstract appeared in *Proceedings of POPL'97*, pages 228–241.

[BSS87] Ed Brinksma, Giuseppe Scollo, and Chris Steenbergen. LOTOS specifications, their implementations and their tests. In *Protocol Specification, Testing and Verification VI (IFIP)*, pages 349–360. North-Holland, 1987.

[BW90] Jos C. M. Baeten and Peter Weijland. *Process Algebra*, volume 18 of *Cambridge Tracts in Computer Science*. Cambridge University Press, 1990.

[CFG99] Paolo Ciancarini, Alessandro Fantechi, and Roberto Gorrieri, editors. *3rd IFIP Conference on Formal Methods for Open Object-based Distributed Systems, Florence, Italy*. Kluwer Academic Publishers, 1999.

[CH93] Søren Christensen and Hans Hüttel. Decidability issues for infinite-state processes—a survey. *Bulletin of the European Association for Theoretical Computer Science (EATCS)*, 51:156–166, 1993.

[CHM93a] Søren Christensen, Yoram Hirshfeld, and Faron Moller. Bisimulation equivalence is decidable for basic parallel processes. In Best [Bes93], pages 143–157.

[CHM93b] Søren Christensen, Yoram Hirshfeld, and Faron Moller. Decomposability, decidability and axiomatisability for bisimulation equivalence on basic parallel processes. In *Proceedings of LICS'93*, pages 386–396. IEEE, Computer Society Press, 1993.

[Chr93] Søren Christensen. *Decidability and Decomposition in Process Algebras.* PhD thesis, Laboratory for Foundations of Computer Science, University of Edinburgh, U. K., 1993.

[CHS95] Søren Christensen, Hans Hüttel, and Colin Stirling. Bisimulation equivalence is decidable for all context-free processes. *Journal of Information and Computation*, 121(2):143–148, 1995.

[Col97] Jean-Louis Colaço. *Analyses Statiques d'un calcul d'acteurs par typage.* Thèse d'État, Institut National Polytechnique de Toulouse, France, 1997.

[CPDS99] Jean-Louis Colaço, Mark Pantel, Fabien Dagnat, and Patrick Sallé. Safety analysis for non-uniform service availability in actors. In Ciancarini et al. [CFG99].

[CPS97] Jean-Louis Colaço, Mark Pantel, and Patrick Sallé. A set constraint-based analyses of actors. In Bowman and Derrick [BD97].

[CSS93] José F. Costa, Amílcar Sernadas, and Cristina Sernadas. Data encapsulation and modularity: Three views of inheritance. In A. Borzyszkowski and S. Sokolowski, editors, *Mathematical Foundations of Computer Science 93*, volume 711 of *Lecture Notes in Computer Science*, pages 382–391. Springer-Verlag, 1993.

[CSS94] José F. Costa, Amílcar Sernadas, and Cristina Sernadas. Object inheritance beyond subtyping. *Acta Informatica*, 31:5–26, 1994.

[CSSE92] José F. Costa, Amílcar Sernadas, Cristina Sernadas, and Hans-Dieter Ehrich. Object interaction. In I. Havel and V. Koubek, editors, *Mathematical Foundations of Computer Science 92*, volume 629 of *Lecture Notes in Computer Science*, pages 200–208. Springer-Verlag, 1992.

[Dal99] Silvano Dal-Zilio. *Calcul bleu: types et objets.* Thèse d'État, Université de Nice - Sophia-Antipolis, France, 1999.

[Dam97] Laurent Dami. Labelled reductions, runtime errors and operational subsumption. In Pierpaolo Degano, Roberto Gorrieri, and Alberto Marchetti-Spaccamela, editors, *Proceedings of ICALP'97*, volume 1256 of *Lecture Notes in Computer Science*, pages 782–793. Springer-Verlag, 1997.

[DF96] Paolo Di Blasio and Kathleen Fisher. A concurrent object calculus. In Montanari and Sassone [MS96], pages 655–670. An extended version appeared as Stanford University Technical Note STAN-CS-TN-96-36, 1996.

[DN67] Ole-Johan Dahl and Kristan Nygaard. Simula begin. Technical report, Norsk Regnesentral (Norwegian Computing Center), Oslo, Norway, 1967.

[EC00] Hans-Dieter Ehrich and Carlos Caleiro. Specifying communication in distributed information systems. *Acta Informatica*, 36:591–616, 2000.

[EGS91] Hans-Dieter Ehrich, J. Goguen, and Amílcar Sernadas. A categorial theory of objects as observed processes. In J. de Bakker, W.-P de Roever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages*, volume 489 of *Lecture Notes in Computer Science*, pages 203–228. Springer-Verlag, 1991.

[ES90] Hans-Dieter Ehrich and Amílcar Sernadas. Algebraic implementation of objects over objects. In J. de Bakker, W.-P de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*, volume 430 of *Lecture Notes in Computer Science*, pages 239–266. Springer-Verlag, 1990.

[FGL+97] Cédric Fournet, Georges Gonthier, Jean-Jacques Lévy, Luc Maranget, and Didier Rémy. A calculus of mobile agents. In Antoni Mazurkiewicz and Józef Winkowski, editors, *Proceedings of CONCUR'97*, volume 1243 of *Lecture Notes in Computer Science*, pages 406–421. Springer-Verlag, 1997.

[Gay93] Simon J. Gay. A sort inference algorithm for the polyadic π-calculus. In *20th Annual Symposium on Principles of Programming Languages (POPL) (Charleston, SC)*. ACM, 1993.

[GH98] Andrew D. Gordon and Paul D. Hankin. A concurrent object calculus: Reduction and typing. In Nestmann and Pierce [NP98].

[GH99] Simon Gay and Malcolm Hole. Types and subtypes for client-server interactions. In *Programming Languages and Systems: Proceedings of the 8th European Symposium on Programming (ESOP'99), Held as Part of the Joint European Conferences on Theory and Practice of Software (ETAPS'99), (Amsterdam, The Netherlands)*, volume 1576 of *Lecture Notes in Computer Science*, pages 74–90. Springer-Verlag, 1999.

[GJS96] James Gosling, Bill Joy, and Guy L. Steele. *The Java Language Specification.* The Java Series. Addison-Wesley, 1996.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

[HBS73] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal, modular actor formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, pages 235–245. William Kaufmann, 1973.

[Hir97] Yoram Hirshfeld. Bisimulation trees and the decidability of weak bisimulations. In *Proceedings of the International Workshop on the Verification of Infinite-State Systems*, volume 5 of *Electronic Notes in Theoretical Computer Science (ENTCS).* Elsevier Science Publishers, 1997.

[HK96] Hans Hüttel and Josva Kleist. Objects as mobile processes. Research Series RS-96-38, BRICS, 1996. Presented at MFPS '96.

[HM85] Matthew Hennessy and Robin Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the ACM*, 32(1):137–161, 1985.

[HMPT94] Constantine Halatsis, Dimitris Maritsas, George Philokyprou, and Sergios Theodoridis, editors. *6th PARLE Conference (Athens, Greece)*, volume 817 of *Lecture Notes in Computer Science.* Springer-Verlag, 1994.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.

[Hon96] Kohei Honda. Composing processes. In Steele [Ste96], pages 344–357.

[HT91] Kohei Honda and Mario Tokoro. An object calculus for asynchronous communication. In Pièrre America, editor, *Proceedings of ECOOP'91*, volume 512 of *Lecture Notes in Computer Science*, pages 133–147. Springer-Verlag, 1991.

[HVK98] Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In Chris Hankin, editor, *Programming Languages and Systems: Proceedings of the 7th European Symposium on Programming (ESOP'98), Held as Part of the Joint European Conferences on Theory and Practice of Software (ETAPS'98), (Lisbon, Portugal)*, volume 1381 of *Lecture Notes in Computer Science*, pages 122–138. Springer-Verlag, 1998.

[HY95] Kohei Honda and Nobuko Yoshida. On reduction-based process semantics. *Theoretical Computer Science*, 152(2):437–486, 1995. An extended abstract appeared in *Proceedings of FST/TCS'93*, volume LNCS 761.

[Kle00] Josva Kleist. *Reasoning about objects using process calculus techniques.* PhD thesis, Department of Computer Science, The Faculty of Engineering and Science, Aalborg University, 2000.

[Kob97] Naoki Kobayashi. A partially deadlock-free typed process calculus. In *Proceedings of LICS'97*, pages 128–139. Computer Society Press, 1997. Extended version as Technical Report 97-02, University of Tokyo, Japan.

[Kob99] Naoki Kobayashi. Quasi-linear types. In Alex Aiken, editor, *Proceedings of POPL'99*, pages 29–42. ACM, 1999.

[Kön99a] Barbara König. *Description and Verification of Mobile Processes with Graph Rewriting Techniques.* PhD thesis, Technische Universität München, 1999.

[Kön99b] Barbara König. Generating type systems for process graphs. In Jos Baeten and Sjouke Mauw, editors, *Proceedings of CONCUR'99*, volume 1664 of *Lecture Notes in Computer Science*, pages 352–367. Springer-Verlag, 1999.

[KPT96] Naoki Kobayashi, Benjamin C. Pierce, and David Turner. Linearity and the π-calculus. In Steele [Ste96], pages 358–371.

[KY95] Naoki Kobayashi and Akinori Yonezawa. Towards foundations of concurrent object-oriented programming—types and language design. *Theory and Practice of Object Systems*, 1(4), 1995. A preliminary summary appeared in *Proceedings of ACM OOPSLA'94*, pages 31-45, under the title Type-Theoretic Foundations for Concurrent Object-Oriented Programming.

[LSV99] Luís Lopes, Fernando Silva, and Vasco T. Vasconcelos. A virtual machine for the TyCO process calculus. In Gopalan Nadathur, editor, *Proceedings of PPDP'99*, volume 1702 of *Lecture Notes in Computer Science*, pages 244–260. Springer-Verlag, 1999.

[LW98] Xinxin Liu and David Walker. Partial confluence of processes and systems of objects. *Theoretical Computer Science*, 206:127–162, 1998. An extended abstract appeared in *Proceedings of TAPSOFT'95* , volume LNCS 915, 217–231.

[Mil89a] Robin Milner. *Communication and Concurrency.* International Series in Computer Science. Prentice Hall, 1989.

[Mil89b] Robin Milner. A complete axiomatisation for observational congruence of finite-state behaviours. *Journal of Information and Computation*, 81(2):227–247, 1989.

[Mil93] Robin Milner. The polyadic π-calculus: A tutorial. In Friedrich L. Bauer, Wilfried Brauer, and Helmut Schwichtenberg, editors, *Logic and Algebra of*

Specification, Proceedings of the International NATO Summer School (Marktoberdorf, Germany, 1991), volume 94 of *Series F*. NATO ASI, Springer-Verlag, 1993. Available as Technical Report ECS-LFCS-91-180, University of Edinburgh, U. K., 1991.

[Mol96] Faron Moller. Infinite results. In Montanari and Sassone [MS96], pages 195–216.

[MPW92] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, part I/II. *Journal of Information and Computation*, 100:1–77, 1992. Available as Technical Reports ECS-LFCS-89-85 and ECS-LFCS-89-86, University of Edinburgh, U. K., 1989.

[MS92] Ugo Montanari and Vladimiro Sassone. Dynamic congruence vs. progressing bisimulation for CCS. *Fundamenta Informaticae*, 16(2):171–199, 1992.

[MS96] Ugo Montanari and Vladimiro Sassone, editors. *7th International Conference on Concurrency Theory (CONCUR), Pisa, Italy*, volume 1119 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[MS98] Massimo Merro and Davide Sangiorgi. On asynchrony in name-passing calculi. In Kim Larsen, Sven Skyum, and Glynn Winskel, editors, *Proceedings of ICALP'98*, volume 1443 of *Lecture Notes in Computer Science*, pages 856–867. Springer-Verlag, 1998.

[ND81] Kristan Nygaard and Ole-Johan Dahl. Simula 67. In R. W. Wexelblat, editor, *History of Programming Languages*. ACM Press, 1981.

[Nie92] Oscar Nierstrasz. Towards an object calculus. In Peter Wegner Mario Tokoro and Oscar Nierstrasz, editors, *Proceedings of the ECOOP'91 Workshop on Object-Based Concurrent Computing*, volume 612 of *Lecture Notes in Computer Science*, pages 1–20. Springer-Verlag, 1992.

[Nie95] Oscar Nierstrasz. Regular types for active objects. In *Object-Oriented Software Composition*, pages 99–121. Prentice Hall, 1995.

[NN96] Flemming Nielson and Hanne R. Nielson. From CML to its process algebra. *Theoretical Computer Science*, 155:179–219, 1996. An extended abstract appeared in *Proceedings of CONCUR'93*, volume LNCS 1243, 493–508.

[NN97] Elie Najm and Abdelkrim Nimour. A calculus of object bindings. In Bowman and Derrick [BD97].

[NNS99a] Elie Najm, Abdelkrim Nimour, and Jean-Bernard Stefani. Guaranteeing liveness in an object calculus through behavioral typing. In *Proceedings of FORTE/PSTV'99*. Kluwer Academic Publishers, 1999.

[NNS99b] Elie Najm, Abdelkrim Nimour, and Jean-Bernard Stefani. Infinite types for distributed objects interfaces. In Ciancarini et al. [CFG99].

[NP98] Uwe Nestmann and Benjamin C. Pierce, editors. *3rd International Workshop on High-Level Concurrent Languages, Nice, France*, volume 16 (3) of *Electronic Notes in Theoretical Computer Science (ENTCS)*. Elsevier Science Publishers, 1998.

[NR00] Uwe Nestmann and António Ravara. What's TyCO, after all? Short abstract, presented at EXPRESS'00, sattelite event of CONCUR'00, 2000.

[NS97] Elie Najm and Jean-Bernard Stefani. Computational models for open distributed systems. In Bowman and Derrick [BD97], pages 157–176.

[Par00] Joachim Parrow. An introduction to the π-calculus. In Alban Ponse Jan A. Bergstra and Gert Smolka, editors, *Handbook of Process Algebra*. Elsevier, 2000. In print.

[PS96] Benjamin C. Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science*, 6(5):409–454, 1996. An extended abstract appeared in *Proceedings of LICS'93*: 376–385.

[Pun96] Franz Puntigam. Types for active objects based on trace semantics. In *IFIP Conference on Formal Methods for Open Object-based Distributed Systems*. Chapman & Hall, 1996.

[Pun97] Franz Puntigam. Coordination requirements expressed in types for active objects. In Memet Aksit and Satoshi Matsuoka, editors, *Proceedings of ECOOP'97*, volume 1241 of *Lecture Notes in Computer Science*, pages 367–388. Springer-Verlag, 1997.

[Pun99] Franz Puntigam. Non-regular process types. In *European Conference on Parallel Processing (Euro-Par)*, volume 1685 of *Lecture Notes in Computer Science*, pages 1334–1343. Springer-Verlag, 1999.

[PW98] Anna Philippou and David Walker. On transformations of concurrent-object programs. *Theoretical Computer Science*, 195(2):259–289, 1998. An extract abstract appeared in *Proceedings of CONCUR'96*, LNCS 1119, 131–146.

[RL99] António Ravara and Luís Lopes. Programming and implementation issues in non-unifom TyCO. Technical report, Department of Computer Science, Faculty of Sciences, University of Porto, 4150 Porto, Portugal, 1999. Presented at the *Workshop on Object-Oriented Specification Techniques for Distributed Systems and Behaviours (OOSDS'99)*. Satellite event of the *1st Conference on Principles, Logics and Implementations of high-level programming languages (PLI'99)*. URL: http://www.tec.informatik.uni-rostock.de/IuK/congr/oosds99/program.htm.

[RRV98] António Ravara, Pedro Resende, and Vasco T. Vasconcelos. Towards an algebra of dynamic object types. In *Semantics of Objects as Processes (SOAP)*, volume NS-98-5 of *BRICS Notes Series*, pages 25–30. Danish Institute of Basic Research on Computer Science (BRICS), 1998.

[RV97] António Ravara and Vasco T. Vasconcelos. Behavioural types for a calculus of concurrent objects. In Christian Lengauer, Martin Griebl, and Sergej Gorlatch, editors, *3th International Euro-Par Conference*, number 1300 in Lecture Notes in Computer Science, pages 554–561. Springer-Verlag, 1997. Extended version available as DM-IST Research Report 06/97.

[San98] Davide Sangiorgi. An interpretation of typed objects into typed π-calculus. *Journal of Information and Computation*, 143(1):34–73, 1998. Earlier version published as Rapport de Recherche RR-3000, INRIA Sophia-Antipolis, 1996.

[San99a] Davide Sangiorgi. The name discipline of uniform receptiveness. *Theoretical Computer Science*, 221(1–2):457–493, 1999. An extended abstract appeared in the *Proceedings of ICALP'97*, volume LNCS 1256, pages 303–313.

[San99b] Davide Sangiorgi. The typed π-calculus at work: A proof of Jones's parallelisation theorem on concurrent objects. *Theory and Practice of Object Systems*, 5(1), 1999. An early version was included in the *Informal proceedings of FOOL 4*, 1997.

[SE91] Amílcar Sernadas and Hans-Dieter Ehrich. What is an object, after all? In R. Meersman, W. Kent, and S. Khosla, editors, *Object Oriented Databases: Analysis, Design and Construction*, pages 39–69. North-Holland, 1991.

[SEC90] Amílcar Sernadas, Hans-Dieter Ehrich, and José F. Costa. From processes to objects. *The INESC Journal of Research and Development*, 1(1):7–27, 1990.

[SFSE89] Amílcar Sernadas, José Fiadeiro, Cristina Sernadas, and Hans-Dieter Ehrich. Abstract object types: A temporal perspective. In B. Banieqbal, H. Barringer, and A. Pnueli, editors, *Temporal Logic in Specification*, volume 398 of *Lecture Notes in Computer Science*, pages 324–350. Springer-Verlag, 1989.

[SK98] Eijiro Sumii and Naoki Kobayashi. A generalized deadlock-free process calculus. In Nestmann and Pierce [NP98].

[SSC95] Amílcar Sernadas, Cristina Sernadas, and José F. Costa. Object specification logic. *Journal of Logic and Computation*, 5(5):603–630, 1995.

[SSC98] Amílcar Sernadas, Cristina Sernadas, and Carlos Caleiro. Denotational semantics of object specification. *Acta Informatica*, 35:729–773, 1998.

[SSE87] Amílcar Sernadas, Cristina Sernadas, and Hans-Dieter Ehrich. Object-oriented specification of databases: an algebraic approach. In P. Hammersley, editor, *Very Large Data Bases 87*, pages 107–116. Morgan and Kaufmann, 1987.

[Ste96] Jr. Guy Steele, editor. *23rd Annual Symposium on Principles of Programming Languages (POPL), St. Petersburg Beach, Florida.* ACM, 1996.

[Str91] Bjarne Stroustrup. *The C++ Programming Language, 2nd. Ed.* Addison-Wesley, 2 edition, 1991.

[Stř98] Jitka Stříbrná. Hardness results for weak bisimilarity of simple process algebras. In *Proceedings of MFCS'98 Workshop on Concurrency*, volume 18 of *Electronic Notes in Theoretical Computer Science (ENTCS).* Elsevier Science Publishers, 1998.

[THK94] Kaku Takeuchi, Kohei Honda, and Makoto Kubo. An interaction-based language and its typing system. In Halatsis et al. [HMPT94], pages 398–413.

[TJ94] Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. *Journal of Information and Computation*, 111(2):245–296, 1994. An extended abstract appeared in *Proceedings of LICS'92*: 162–173.

[Tur96] David N. Turner. *The Polymorphic Pi-Calculus: Theory and Implementation.* PhD thesis, Laboratory for Foundations of Computer Science, University of Edinburgh, U. K., 1996. CST-126-96 (also published as ECS-LFCS-96-345).

[Vaa86] Frits Vaandrager. Process algebra semantics for POOL. Report CS-R862, Centre for Mathematics and Computer Science, Amsterdam, The Netherlands, 1986.

[Vas94a] Vasco T. Vasconcelos. *A process-calculus approach to typed concurrent objects.* PhD thesis, Keio University, Japan, 1994.

[Vas94b] Vasco Thudichum Vasconcelos. Predicative polymorphism in π-calculus. In Halatsis et al. [HMPT94], pages 425–437.

[Vas99] Vasco T. Vasconcelos. Processes, functions, datatypes. *Theory and Practice of Object Systems*, 5(2):97–110, 1999.

[vG93] Rob J. van Glabbeek. The linear time—branching time spectrum II (the semantics of sequential systems with silent moves). In Best [Bes93], pages 66–80.

[VH93] Vasco T. Vasconcelos and Kohei Honda. Principal typing schemes in a polyadic π-calculus. In Best [Bes93], pages 524–538.

[VR99] Vasco T. Vasconcelos and António Ravara. Comunication errors in the pi-calculus are undecidable. *Information Processing Letters*, 71(5–6):229–233, 1999.

[VT93] Vasco T. Vasconcelos and Mario Tokoro. A typing system for a calculus of objects. In *Proceedings of the 1st International Symposium on Object Technologies for Advanced Software*, volume 742 of *Lecture Notes in Computer Science*, pages 460–474. Springer-Verlag, 1993.

[Wal95] David Walker. Objects in the π-calculus. *Journal of Information and Computation*, 116(2):253–271, 1995.

[Weg90] Peter Wegner. Concepts and paradigms of object-oriented programming. *OOPS Messenger*, 1(1):7–87, 1990. Expansion of OOPSLA'89 Keynote Talk.

[WZ88] Peter Wegner and Stanley B. Zdonik. Inheritance as an incremental modification mechanism or what like is and isn't like. In *Proceedings of the 2nd European Conference on Object-Oriented Programming (ECOOP), Oslo, Norway*, volume 322 of *Lecture Notes in Computer Science*, pages 55–77. Springer-Verlag, 1988.

[Yos96] Nobuko Yoshida. Graph types for monadic mobile processes. In *Proceedings of the 16th Conference on Foundations of Software Technology and Theoretical Computer Science (FST/TCS)*, volume 1180 of *Lecture Notes in Computer Science*, pages 371–386. Springer-Verlag, 1996. Extended version as Technical Report ECS-LFCS-96-350, University of Edinburgh.

[ZP97] Xiaogang Zhang and John Potter. Class-based models in the pi-calculus. In Christine Mingins, Roger Duke, and Bertrand Meyer, editors, *Proceeding of the 25th International Conference in Technology of Object-Oriented Languages and Systems (TOOLS Pacific'97, Melbourne, Australia)*, pages 219–231, 1997.