D I S S E R T A T I O N

Expressiveness and Applicability of Process Types

ausgeführt zum Zwecke der Erlangung des akademischen Grades
eines Doktors der technischen Wissenschaften unter der Leitung von

O. Univ.-Prof. Dipl.-Ing. Dr. Manfred Brockhaus
E 185
Institut für Computersprachen

eingereicht an der Technischen Universität Wien
Technisch-Naturwissenschaftliche Fakultät

von

Christof Peter
9126568
Kaiser-Franz-Josef-Straße 133
6845 Hohenems

Wien, am 7. Februar 2000

Kurzfassung

In dieser Dissertation werden verteilte Systeme basierend auf Objektkalkülen behandelt. Objekte in diesen Kalkülen kommunizieren, indem sie Nachrichten versenden. Statisch überprüfte Prozeßtypen garantieren, daß jede gesendete Nachricht vom Empfänger verstanden wird. Ein Prozeßtyp beschreibt das Verhalten eines Objekts in Abhängigkeit seines Zustands, der von den bisher empfangenen Nachrichten abhängt.

Der Prozeßtypkalkül hat zwei wichtige Eigenschaften: Jedes Objekt hat eine eindeutige Adresse, und die logische Reihenfolge von Nachrichten bleibt erhalten. Während die erste Eigenschaft in Kalkülen, die auf dem Actor-Modell basieren, verbreitet ist, wird die zweite von den meisten asynchronen Kalkülen nicht garantiert.

Diese Arbeit untersucht die Eigenschaften von Prozeßtypen und des ihnen zugrundeliegenden Kalküls im Vergleich zu anderen Kalkülen und deren Typsystemen. Insbesondere werden sie mit Vasconcelos TyCO-Kalkül, einem Kalkül, der auch Konzepte von Milners π-Kalkül reflektiert, verglichen. Eine Übersetzung von untypisierten und typisierten TyCO-Programmen in den (untypisierten bzw. typisierten) Prozeßtypkalkül wird vorgestellt. Untypisierte Programme des Prozeßtypkalküls können in untypisierte TyCO-Programme übersetzt werden. Es ist nicht möglich, alle typisierten Programme des Prozeßtypkalküls in TyCO-Programme zu übersetzen, da Prozeßtypen ausdrucksstärker sind und die Untertyp-Relation unterstützen.

Anschließend werden Prozeßtypen höherer Ordnung präsentiert und auf ihnen eine "Matching"-Relation definiert. Es wird ein Typüberprüfungsalgorithmus definiert und bewiesen, daß er richtig und vollständig ist.

Abschließend wird skizziert, wie eine (dynamische) Version von Prozeßtypen in eine praktische Anwendung wie CORBA integriert werden kann. Die CORBA-IDL wird mit Prozeßtypen erweitert. Eine Typüberprüfung wird in den CORBA-IDL-Übersetzer integriert und jedesmal aktiviert, wenn eine Methode aufgerufen wird. Alle Nachrichten, die nicht verstanden werden könnten, werden entfernt anstatt gesendet.

Abstract

In this thesis we deal with distributed systems based on calculi of objects that communicate by sending messages. Statically checked process types ensure that each sent message is understood by the receiver. A process type describes also behavior depending on the object's state. The state of an object depends on the previously received messages.

The process type calculus has two important properties: Each object has a unique identifier, and the logical ordering of messages is preserved. While the first property is common in calculi related to the actors model, the second property is not explicitly ensured in most asynchronous calculi.

We examine the properties of process types and the underlying calculus in comparison to other calculi and their type systems. Especially, we compare them with Vasconcelos' TyCO calculus, a calculus which contains the concepts of Milner's π-calculus. We give a fully abstract encoding of untyped TyCO programs to our calculus and vice versa. We also give a fully abstract encoding of typed TyCO programs into our calculus with types. But it is not possible to encode all typed programs in the process type calculus in TyCO; process types are more expressive than TyCO types and support subtyping.

Then we propose higher order process types and a matching relation on them. We present a type checking algorithm and prove that it is sound and complete.

Finally, we describe how a (dynamic) version of process types can be integrated in a practical application like CORBA. The CORBA Interface Definition Language (IDL) is extended with process types. A process type checker is integrated into the CORBA-IDL compiler and activated on each service invocation. All messages that may not be understood by the receiver are discarded instead of being sent.

Contents

Chapter 1

Introduction

Software is becoming a more and more complex collection of distributed, interacting components. A common approach in software development is to use object-oriented technologies for the design and specification of components [Weg90, Boo94, Car93]. These technologies support re-use, maintenance, separate compilation and well-known design principles and patterns [GHJV95]. Strong and static typing are reliable means to support simple and expressive design [CW85]. A type specifies a contract between an object and its users [Mey94].

The communication between objects is often based on message passing. A message corresponds to the invocation of a service. Communication protocols able to manage the complexity of communication between the objects are needed. Most protocols describe the available services that an object offers, regardless to the state of the object which may change due to interaction [Car93, Mey93].

In many cases the services an object offers depend on the state of the object [Nie93, LL99, NN99, RV97, CPS97]. We use process types [Pun96b, Pun96a, Pun97a, PP98, Pun98, Pun99, PP99a] to describe protocol services whose availability is non-uniform. Process types can be applied to ensure that other objects always have consistent state information. Process types are very expressive, but they require a calculus that was designed to support them.

In this thesis, we examine the principles of the calculus underlying process types. To achieve this purpose, we start with a very simple core version of the calculus which supports only the basic primitives. It is compared with other calculi and, after pointing out the differences, we give a fully abstract encoding from our calculus to the TyCO calculus [Vas99]. This calculus was chosen as representative for channel based calculi like the π-calculus [MPW92, Mil92]. The techniques used for encoding our calculus to TyCO could also be used to translate our calculus to the π-calculus; but this is not necessary, because there is a fully abstract encoding from TyCO to the π-calculus. Then, we show that the process type system for the core calculus is more expressive than the type-system used in TyCO.

We extend the calculus with more expressive primitives supporting genericity, subtyping and subsumption. We give a type checking algorithm and examine equivalence relations on types. Then we take a look at approaches for higher order subtyping and show how these higher order types can be integrated in our calculus. Again, we give a checking algorithm for the higher order types.

Finally, we want to show that our concept is also practically applicable. The CORBA Interface Definition Language (IDL) [OMG98a] is a standard for integrating objects written in different languages using different paradigms. The IDL describes the services an object[1] offers. There is a demand for a more expressible and flexible IDL. Various proposals for additional specifications in the IDL were presented [Siv95, LC93]. We integrate process types in the IDL, a simple but expressive extension. The concepts of process types support specifications of non-uniform interfaces.

This thesis is structured in the following way: Each chapter starts with a short introduction and an outline of the chapter. At the end of the chapter, there is a discussion of the presented work in the context of related work and different approaches.

In the first part we explore type concepts for distributed systems and the calculi they are based on. In the first chapter of this part we will examine the π-calculus which is the most canonical calculus for distributed system. Its only entities are names, and communication is based on passing of names via named channels. We will examine the power and limitations of type systems designed for such calculi.

The second chapter of the first part examines various other approaches. The main part of this chapter deals with the TyCO calculus which is closely related to the π-calculus. It supports some higher-level constructions like abstractions, and supports type inference. A different, more powerful type system able to handle behavioral types in TyCO is also examined. An approach which relates closely to our process types is given at the end of this part.

The next part deals with the concepts of process types. First of all we take a close look at the underlying core calculus without types. We show that this calculus has the same expressiveness as the TyCO calculus when types are neglected. When we also look at types for the process type calculus, it becomes obvious that process types are more expressive than the standard types in TyCO.

A powerful extension for process types supporting genericity and dynamic comparisons on types is also presented. We present a concept for subtyping and in the last chapter of this part we examine an extension of the type concept to higher order subtyping.

In the last part we examine how process types can be integrated in a standard system like the Common Object Broker Architecture (CORBA). An extension of the IDL is presented and a description of the changes that are needed to check this extension dynamically is given.

[1] If the language is not object-based, the entities corresponding to objects are wrapped into CORBA-objects.

Part I

Typed Concurrent Calculi

Chapter 2

Types for the π-Calculus

2.1 Introduction

The π-calculus [MPW92, Mil92] is a development from CCS [Mil89] proposed by Milner. The π-calculus supports mobility by allowing names to be communicated. The original calculus was based on monadic synchronous communication. Since then many variants of the π-calculus have been proposed, for example polyadic [PS93, Yos96, QW98], asynchronous [HT91, Bou92] and polymorphic [Tur92] versions. The π-calculus is untyped. But, soon many type-systems (also called sortings) for the polyadic π-calculus were proposed, for example by Milner, Gay, Vasconcelos and Honda and Boudol [Mil92, Gay93, VH93, PS93, Yos96, QW98, Bou97].

The remainder of this chapter is structured as follows: In Section 2.2, we shortly sketch syntax and semantics of the monadic π-calculus. In Section 2.3 on page 11 we discuss the polyadic π-calculus, and explain the relation to the monadic π-calculus, then we give some standard typing rules. We present a more complex typing scheme proposed by Pierce and Sangiorgi in Section 2.4 on page 14. They proposed a subtyping relation, but they failed to express sequences of messages. We take a look at solutions of this problem in the following Section 2.5 on page 18, where two approaches to express sequences of messages in the context of translating the (typed) polyadic π-calculus to the (typed) monadic π-calculus are examined. The chapter is finished with a discussion of the concepts in Section 2.6 on page 23.

2.2 The Monadic π-Calculus

The computation of the π-calculus is based on naming of channels [MPW92, ACS96]. The basic primitive entities are *names* $a, b, \ldots, x, y, \ldots \in \mathcal{X}$ which have no structure. The only entity is a *process* $P, Q, \ldots \in \mathcal{P}$ with the following syntax:

$$P ::= \Sigma_{i \in I} \pi . P_i \quad | \quad P|Q \quad | \quad !P \quad | \quad (\nu x)P \quad | \quad \mathbf{0}$$

where I is a finite indexing set. The prefix π is of the form

$$\pi ::= x(y) \quad | \quad \overline{x}\langle y \rangle$$

The first prefix binds a name y in the prefixed process: A name called y is input from channel x. The second form means that a name y is output to the channel x. A name always refers to a channel. The name x is used for input while the *co-name* $\overline{x}$ is used for output.

The summation $\Sigma_{i\in I}\pi.P_i \equiv P_1 + P_2 + \cdots + P_n$ represents a process able to take part in one of several alternatives for communication. The choice is made depending on the input or output; the process can never commit to one alternative until it occurs, and this occurrence precludes the other alternatives.

$P|Q$ means that P and Q are concurrently and independently active. The *replication* operator $!P$ means $P|P|\ldots$. The restriction $(\nu x)P$ restricts the use of the name x to P. This means that a new name x is declared, distinct from all external names, for use in P. The process $\mathbf{0}$ is an empty (nil) process. We will abbreviate processes like $x(y).\mathbf{0}$ to $x(y)$. Restrictions of the form $(\nu x)(\nu y)P$ are abbreviated to $(\nu xy)P$.

2.2.1 Examples

Consider the process

$$\overline{x}\langle y\rangle \mid x(u).\overline{u}\langle v\rangle \mid \overline{x}\langle z\rangle$$

Only one of two communications can occur along the channel x. The result may be

$$\mathbf{0} \mid \overline{y}\langle v\rangle \mid \overline{x}\langle z\rangle \qquad \text{or} \qquad \overline{x}\langle y\rangle \mid \overline{z}\langle v\rangle \mid \mathbf{0}$$

Now consider a process

$$(\nu x)(\overline{x}\langle y\rangle \mid x(u).\overline{u}\langle v\rangle) \mid \overline{x}\langle z\rangle.$$

The free x in the process at the very right is different from the bound x within the braces. The only communication that can happen results in

$$\mathbf{0} \mid \overline{y}\langle v\rangle \mid \overline{x}\langle z\rangle$$

Another variant is

$$\overline{x}\langle y\rangle \mid !x(u).\overline{u}\langle v\rangle \mid \overline{x}\langle z\rangle$$

The part in the middle can spin off one copy to communicate with the process on the left; The system becomes

$$\mathbf{0} \mid \overline{y}\langle v\rangle \mid !x(u).\overline{u}\langle v\rangle \mid \overline{x}\langle z\rangle$$

Then it can spin off another copy to communicate with the right process:

$$\mathbf{0} \mid \overline{y}\langle v\rangle \mid !x(u).\overline{u}\langle v\rangle \mid \overline{z}\langle v\rangle \mid \mathbf{0}$$

$$
\begin{array}{rlr}
P|Q \equiv_\pi Q|P & & (\textsc{Commut}) \\
(P|Q)|R \equiv_\pi P|(Q|R) & & (\textsc{Assoc}) \\
P|\mathbf{0} \equiv_\pi P & & (\textsc{Zero}) \\
!P \equiv_\pi P|!P & & (\textsc{Repl}) \\
((\nu x)P)|Q \equiv_\pi ((\nu x)P|Q) & \text{if } x \notin \mathit{fn}(Q) & (\textsc{Extrusion}) \\
(\nu x)(\nu y)P \equiv_\pi (\nu y)(\nu x)P & & (\textsc{Switch}) \\
(\nu x)\mathbf{0} \equiv_\pi \mathbf{0} & & (\textsc{Drop})
\end{array}
$$

Figure 2.1: Structural congruence in the π-calculus

2.2.2 Operational Semantics

Definition 1 (Free names) *We define the set of free names in P, denoted $\mathit{fn}(P)$, inductively:*

$$
\begin{array}{ll}
\mathit{fn}(\mathbf{0}) = \emptyset & \mathit{fn}(\overline{a}\langle x\rangle.P) = \{a\} \cup \{x\} \cup \mathit{fn}(P) \\
\mathit{fn}(a(y).P) = \{a\} \cup (\mathit{fn}(P) \setminus \{y\}) & \mathit{fn}(!P) = \mathit{fn}(P) \\
\mathit{fn}((\nu x)P) = \mathit{fn}(P) \setminus \{x\} & \mathit{fn}(P|Q) = \mathit{fn}(P) \cup \mathit{fn}(Q)
\end{array}
$$

Definition 2 (Structural congruence) *Structural congruence $\equiv_\pi$ is the smallest congruence relation over $\mathcal{P}$ such that the rules in Figure 2.1 together with α-conversion hold.*

Concurrency of processes is commutative and associative with $\mathbf{0}$ as neutral element. The replication of a process produces an arbitrary number of copies of this process in parallel. The ordering of scope restrictions is changeable and a restriction can be extended if the restricted name is not needed by another process.

Definition 3 (Reduction) *One step reduction in the π-calculus is the smallest relation generated by the rules in Figure 2.2 on the following page. The transitive and reflexive closure of $\rightarrow$ is written $\xrightarrow{*}$.*

The most important rule is Rule (Comm). When a process sends a name z via a channel x and another process wants to receive a name (which this process calls y) via $\overline{x}$, then z substitutes y in the second process. Concurrent processes work independently. The reduction rules can be applied to them in any arbitrary ordering. Restrictions of names on processes remain unchanged by reduction. Structural equivalence may be applied on a process before and after reduction rules are applied.

2.2.3 Barbed Bisimulation

The most popular way of defining behavioral equivalences on processes is via the notion of bisimulation. Barbed bisimulation [MS92] is a tool for uniformly defining bisimulation-based equivalences in different calculi. For our work we need a special version based on communication barbs. It detects for each output name $\overline{a}$ the possibility of performing communication with the external environment along $\overline{a}$.

$$x(y).P|\overline{x}\langle z\rangle.Q \to P|Q \qquad (\textsc{Comm})$$

$$\frac{P \to P'}{P|Q \to P'|Q} \qquad (\textsc{Par})$$

$$\frac{P \to P'}{(\nu x)P \to (\nu x)P'} \qquad (\textsc{Res})$$

$$\frac{P \equiv_\pi P' \quad P' \to Q' \quad Q' \equiv_\pi Q}{P \to Q} \qquad (\textsc{Struct})$$

Figure 2.2: Reduction rules in the monadic π-calculus

Definition 4 (Commitment) *The strong commitment of a process on a channel expresses the fact that the process is ready to send messages on that channel. Formally, $P \downarrow_{\overline{a}}$ if P can make an output action whose subject is a, that is there exist P' and b such that $P \to P'$ using $\overline{a}\langle b\rangle$. The weak commitment is defined as:*

$$P \Downarrow_{\overline{a}} \quad \textit{if} \quad P \xrightarrow{*} P' \quad \textit{and} \quad P' \downarrow_{\overline{a}}$$

For example, if $P \stackrel{\text{def}}{=} (\nu c)(\overline{c}\langle b\rangle|a(x).d(y))$, then $P \downarrow_{a}$, but not $P \downarrow_{\overline{c}}$ or $P \downarrow_{\overline{b}}$, or $P \downarrow_{\overline{d}}$.

From the definition of reduction and commitment, the definition of bisimulation is straight forward:

Definition 5 (Barbed bisimulation) *A symmetric relation S on terms is a (strong) barbed bisimulation if for all $(P,Q) \in S$ the following holds:*

1. *If $P \downarrow_{\overline{a}}$ then $Q \downarrow_{\overline{a}}$.*

2. *If $P \to P'$ then $Q \to Q'$ and $(P', Q') \in S$.*

We denote $\stackrel{\bullet}{\sim}$ by the largest barbed bisimulation. The notation of weak barbed simulation is obtained by replacing everywhere the commitment $\downarrow$ with $\Downarrow$ and the transition $\to$ with $\to^*$. We denote the largest weak barbed bisimulation with $\stackrel{\bullet}{\approx}$.

The barbed bisimulation shall be refined to an equivalence that also respects parallel composition [MS92, PS93].

Definition 6 (Barbed equivalence) *The relations of strong and weak barbed equivalence are defined as follows:*

$$P \sim_b Q \ \textit{if} \ \forall R(P|R \stackrel{\bullet}{\sim} Q|R)$$
$$P \approx_b Q \ \textit{if} \ \forall R(P|R \stackrel{\bullet}{\approx} Q|R)$$

2.3 The Polyadic π-Calculus

It may be useful to use multiple inputs and outputs along the same name. To avoid mix-ups if the information is sent sequentially and read from more than one reader, a restricted name for communication shall be used. The following abbreviations are defined [Mil92]:

$$\begin{aligned} x(z_1, \ldots, z_n) &= x(w).w(z_1). \ldots .w(z_n) \\ \overline{x}\langle a_1, \ldots, a_n\rangle.Q &= (\nu w)\overline{x}\langle w\rangle.\overline{w}\langle a_1\rangle. \ldots .\overline{w}\langle a_n\rangle \end{aligned}$$

The notion of *substitution* of free occurrences of x by z in P, denoted $P[z/x]$, is defined in the usual way, and so is α-conversion (renaming of bound names).

For each meta-symbol e, a sequence $e_1, \ldots, e_n$ can be abbreviated by $\tilde{e}$ if n is arbitrary. Likewise, $\{\tilde{e}\}$ denotes the set containing $e_1, \ldots, e_n$, and $|\tilde{e}|$ the length of the sequence. For each operator $\circ$, an expression $\tilde{e} \circ \tilde{g}$ (with $|\tilde{e}| = |\tilde{g}|$) stands for $e_1 \circ g_1, \ldots, e_i \circ g_n$. The simultaneous substitution of e_i for all free occurrences of g_i (with $i = 1, \ldots, |\tilde{e}|$) in f is denoted by $f[\tilde{e}/\tilde{g}]$, where $\tilde{e}$ and $\tilde{g}$ are sequences of the same length, and $\tilde{g}$ is a list of pairwise different names.

The process equivalence rules are the same as in the monadic π-calculus. The reduction rules are also the same, only the rule for communication is changed:

$$\frac{m = n}{a(b_1, \ldots, b_m).P|\overline{a}\langle c_1, \ldots, c_n\rangle.Q \to P[c_1, \ldots, c_n/b_1, \ldots, b_n]|Q} \qquad (\textsc{Comm})$$

For example, the expressions

$$\begin{aligned} P_t &\stackrel{\text{def}}{=} b(t, f).\overline{t}\langle\rangle.P_t \\ P_f &\stackrel{\text{def}}{=} b(t, f).\overline{f}\langle\rangle.P_f \end{aligned}$$

denote processes that accept a pair of names, t and f form a channel b, responds by sending a null tuple along t or f and returns to its original state. These processes can be used as encodings of the boolean values *true* and *false*: To test whether the process waiting for input on b is P_t or P_f, it suffices to put it in parallel with the process

$$\textsc{Test} \stackrel{\text{def}}{=} (\nu x\, y)(\overline{b}\langle x, y\rangle.\mathbf{0}|x().Q_t|y().Q_f)$$

which creates a pair of channels with the new names x and y and sends them along b. Then it waits for a response along x or y before continuing with Q_t or Q_f. If Test is run in parallel with P_t then Q_t is activated; If P_f is run in parallel, Q_f is activated.

The following example is a simplified version of a system used by Orava and Parrow [OP92], presented by Milner in [Mil92]. A center is in permanent contact with two base stations. A car with a mobile phone moves and should always be in contact with a base. If it gets too far from its current base station, a hand-over is initiated (in an unspecified way) and the car gets in contact with the other base and looses contact to the previous one.

The system in the state where the CAR is in contact with BASE_1 is expressed by the following process:

$$\begin{aligned}\text{SYSTEM}_1 \stackrel{\text{def}}{=} & (\nu\, talk_1\; talk_2\; switch_1\; switch_2\; give_1\; give_2\; alert_1\; alert_2)\\ & (\text{CAR}(talk_1, switch_1)|\text{BASE}_1|\text{IDLEBASE}_2|\text{CENTRE}_1)\end{aligned}$$

A CAR is parametric upon a *talk* channel and a *switch* channel. The *talk* channel can be used repeatedly. Along *switch* it may receive two new channels:

$$\text{CAR}(talk, switch) \stackrel{\text{def}}{=} talk().\text{CAR}(talk, switch) + switch(talk', switch').\text{CAR}(talk', switch')$$

A BASE can talk repeatedly with the CAR; but at any time it can receive along its *give* channel two new channels through which it should communicate to the CAR and then become idle itself. There are two bases, we use $i \in 1, 2$:

$$\begin{aligned}&\text{BASE}_i(talk_i, switch_i, give_i, alert_i) \stackrel{\text{def}}{=} talk_i.\text{BASE}_i(talk_i, switch_i, give_i, alert_i) +\\ &\quad give_i(talk'_i, switch'_i).\overline{switch'_i}\langle talk'_i, switch'_i\rangle.\text{IDLEBASE}_i(talk_i, switch_i, give_i, alert_i)\end{aligned}$$

An IDLEBASE may be told on its *alert* channel to become active:

$$\text{IDLEBASE}_i(talk_i, switch_i, give_i, alert_i) \stackrel{\text{def}}{=} alert_i.\text{BASE}_i(talk_i, switch_i, give_i, alert_i)$$

The CENTRE_1 which initially knows that the car is in contact with BASE_1, can decide to transmit the channels $talk_2$ and $switch_2$ to the CAR via BASE_1, and alert BASE_2 of this fact.

$$\begin{aligned}\text{CENTRE}_1 &\stackrel{\text{def}}{=} \overline{give_1}\langle talk_2, switch_2\rangle.alert_2.\text{CENTRE}_2\\ \text{CENTRE}_2 &\stackrel{\text{def}}{=} \overline{give_2}\langle talk_1, switch_1\rangle.alert_1.\text{CENTRE}_1\end{aligned}$$

2.3.1 Translating the Polyadic to the Monadic π-Calculus

The atomic communication of tuples of names is expressible in the monadic calculus [MPW92]. We inductively define a translation:

Definition 7 (Translating polyadic to monadic π-calculus) *Translation $[\![\cdot]\!]$ from terms of the polyadic π-calculus to monadic terms is defined as follows:*

$$\begin{aligned}[\![x(z_1,\dots,z_n).P]\!] &= x(w).w(z_1).\dots w(z_n).[\![P]\!]\\ [\![\overline{x}\langle a_1,\dots,a_n\rangle.Q]\!] &= (\nu w)\overline{x}\langle w\rangle.\overline{w}\langle a_1\rangle.\dots\overline{w}\langle a_n\rangle.[\![Q]\!]\end{aligned}$$

in each case w is fresh and $[\![\mathbf{0}]\!] = \mathbf{0}$, $[\![P|P']\!] = [\![P]\!]|[\![P']\!]$, $[\![(\nu v)P]\!] = (\nu v)P$, $[\![!P]\!] = ![\![P]\!]$ and $[\![P + P']\!] = [\![P]\!] + [\![P']\!]$.

The transmission of a tuple $\tilde{a} = a_1, \dots, a_n$ is expressed by

$$\begin{aligned}& [\![x(z_1,\dots,z_n).P|\overline{x}\langle a_1,\dots,a_n\rangle.Q]\!]\\ =\ & [\![x(z_1,\dots,z_n).P]\!]|[\![\overline{x}\langle a_1,\dots,a_n\rangle.Q]\!]\\ \rightarrow\ & (\nu w)(w(z_1).\dots w(z_n).[\![P]\!]|\overline{w}\langle a_1\rangle.\dots\overline{w}\langle a_n\rangle.[\![Q]\!])\\ \rightarrow^n & [\![P]\!][\tilde{a}/\tilde{z}]|[\![Q]\!]\\ =\ & [\![P[\tilde{a}/\tilde{z}]|Q]\!]\end{aligned}$$

where $\tilde{z} = z_1, \ldots, z_n$. The first interaction creates a private name; the subsequent communications transfer the names $a_1, \ldots, a_n$ from sender to receiver via that name. Although communication of an n-tuple is represented using $n+1$ interactions, it can be considered atomic because of

$$(\nu w)(w(z_1).\ldots w(z_n).[\![P]\!] | \overline{w}\langle a_1\rangle.\ldots\overline{w}\langle a_n\rangle.[\![Q]\!]) \sim_b [\![P]\!][\tilde{a}/\tilde{z}] | [\![Q]\!]$$

It is possible to translate terms with summation to π-terms without summation, using a technique presented in [HT91]. A two step protocol is used in which a message first asks the object method names, and then selects one of them. The term

$$a(x_1, \ldots, x_n).P + b(y_1, \ldots, y_m).Q$$

can be translated into

$$(\nu l)(\nu m)(c(r).\overline{r}\langle l_1 l_2\rangle.(l_1(x).P | l_2(y).Q))$$

where c could be regarded as the "name" of the summation. A message of the form

$$\overline{a}\langle v\rangle$$

can be translated into

$$(\nu r)(\overline{c}\langle r\rangle.r(l_1 l_2).\overline{l_1}\langle v_1 \ldots v_n\rangle)$$

In the remainder of this chapter, the summation of the π-calculus is omitted.

2.3.2 Typing Rules for the Polyadic π-Calculus

Milner observed that the trio of processes consisting of P_t, P_f and TEST obey a strict typing discipline and decided to associate typing information only to names [Mil92]. Processes are either well typed under a particular set of assumptions for their bound and free names, or they are not. This typing information is completely static.

The typing[1] rules given in Figure 2.3 on the next page are quite simple. The empty process is always well behaved. A replicated process is well behaved if a single copy of that process is well behaved. The parallel composition of two processes is well behaved if each of them is well behaved in isolation (Par). If a process has a restriction, it is well behaved if the body is well behaved with respect to the new typing introduced by the restriction (Res). The communication rules have to ensure that the number and types of the parameters are correct (In, Out).

The types for boolean values given in Section 2.3 on page 11 reflect that t, f, x and y are used to communicate tuples of arity zero, but b is always used to communicate pairs of names that are used to communicate empty tuples.

$$\begin{array}{rl} t, f, x, y : S_e & \quad S_e \mapsto () \\ b : S_p & \quad S_p \mapsto (S_e, S_e) \end{array}$$

The type information is associated only to channels.

[1] In the π-calculus the term "sorting" is also common. To keep consistent with our nomenclature we use the term "typing" also in the π-calculus.

$$\Gamma \vdash \mathbf{0} \tag{Nil}$$

$$\frac{\Gamma \vdash P}{\Gamma \vdash !P} \tag{Repl}$$

$$\frac{\Gamma \vdash P_1 \quad \Gamma \vdash P_2}{\Gamma \vdash P_1|P_2} \tag{Par}$$

$$\frac{\Gamma, z{:}s \vdash P}{\Gamma \vdash (\nu z)P} \quad z \notin fn(\Gamma) \tag{Res}$$

$$\frac{s \equiv (t_1, \ldots, t_n) \quad \Gamma \vdash P}{\Gamma, x{:}s, y_1{:}t_1, \ldots, y_n{:}t_n \vdash \overline{x}\langle y_1, \ldots, y_n \rangle.P} \tag{Out}$$

$$\frac{s \equiv (t_1, \ldots, t_n) \quad \Gamma, z_1{:}t_1, \ldots, z_n{:}t_n \vdash P}{\Gamma, x{:}s \vdash x(z_1, \ldots, z_n).P} \tag{In}$$

Figure 2.3: Standard typing rules for the polyadic π-calculus

2.4 Pierce and Sangiorgi's Types for the Polyadic π-Calculus

2.4.1 Syntax and Types

The syntax of typed processes is given below, and the syntax of types is given in Figure 2.4 on the next page [PS93].

$$\begin{aligned} P ::= \mathbf{0} \quad | \quad P|Q \quad | \quad (\nu a{:}S)P \quad | \quad a(a_1{:}S_1, \ldots, a_n{:}S_n).P \quad | \\ \overline{a}\langle a_1, \ldots, a_n \rangle.P \quad | \quad !P \quad | \quad \mathit{wrong} \end{aligned}$$

The metavariables $P, Q, R \ldots$ denote process expressions, $a, b, c, \ldots$ denote channels or names, $S, T, U, \ldots$ are used for types, $\Gamma, \Delta, \ldots$ are used for typing assumptions (sometimes abbreviated as typings).

As defined in Section 2.3 on page 11, a process may be a an empty process or a parallel composition of other processes. Names can be restricted with the restrict operator ν and have a typing associated with them. A process $a(a_1{:}S_1, \ldots, a_n{:}S_n).P$ is the input of names $a_1, \ldots, a_n$ with a typing $S_1, \ldots, S_n$ on the channel a; then the process P is executed. An output operation $\overline{a}\langle a_1, \ldots, a_n \rangle.P$ is the sending of the names $a_1, \ldots, a_n$ via the channel a. The explicit replication operator $!P$ stands for the parallel composition of a (countable) infinite number of copies of P. The constant *wrong* stands for a process where a run-time error has occurred, i.e., one in which an attempted communication has involved an arity mismatch or a violation of an I/O restriction.

A typing Γ can either be empty or contain expressions $a{:}S$, where a a is a channel and S is a type associated to it. A channel type can be a tagged tuple $(S_1, \ldots, S_n)^I$, a recursive type $\mu A.S$ or a variable A. The tags I can be r (input only), w (output only) or b (both).

$$
\begin{array}{rcl}
\Gamma & ::= & \emptyset \quad | \quad \Gamma, a{:}S \\
S & ::= & (S_1, \ldots, S_n)^I \quad | \quad \mu A.S \quad | \quad A \\
I & ::= & r \quad | \quad w \quad | \quad b
\end{array}
$$

Figure 2.4: Syntax of π-calculus typings

$$
\begin{array}{rcll}
P|Q & \equiv_\pi & Q|P & \text{(COMMUT)} \\
(P|Q)|R & \equiv_\pi & P|(Q|R) & \text{(ASSOC)} \\
P|\mathbf{0} & \equiv_\pi & P & \text{(ZERO)} \\
!P & \equiv_\pi & P|!P & \text{(REPL)} \\
((\nu a{:}S)P)|Q & \equiv_\pi & ((\nu a{:}S)P|Q) \quad \text{if } a \notin fn(Q) & \text{(EXTRUSION)} \\
(\nu a{:}S)(\nu b{:}T)P & \equiv_\pi & (\nu b{:}T)(\nu a{:}S)P \quad \text{if } a \neq b & \text{(SWITCH)} \\
(\nu a{:}S)\mathbf{0} & \equiv_\pi & \mathbf{0} & \text{(DROP)} \\
P|\, wrong & \equiv_\pi & wrong & \text{(PAR-WRONG)} \\
!wrong & \equiv_\pi & wrong & \text{(REPL-WRONG)} \\
(\nu a{:}S)\, wrong & \equiv_\pi & wrong & \text{(RESTR-WRONG)}
\end{array}
$$

Figure 2.5: Process-equivalence in the typed polyadic π-calculus

2.4.2 Typed Reduction Semantics

The reduction semantics use two relations: A structural equivalence on process terms (Figure 2.5) that permits the rearrangement of parallel compositions, replications and restrictions so that the participants of a potential communication can compared immediately and reduction rules.

The process-equivalence relation in the polyadic π-calculus is similar to the process-equivalence in the monadic π-calculus. New rules are the rules for *wrong*.

The reduction rules in the polyadic π-calculus are analogous to the rules in the monadic π-calculus (See Figure 2.6 on the following page). Different are the rules for communication: A transmission of messages with the wrong (not expected) number of arguments is a communication error.

2.4.3 The Subtyping Relation

Definition 8 (Sub-tag-relation) *The basic sub-tag relation $I \leq J$ is the least preorder containing b$\leq$r and b$\leq$w.*

Definition 9 (I/O-tree) *An I/O-tree is a finitely branching tree whose nodes are labelled with symbols from {r,w,b}. The metavariables $\mathcal{S}$, $\mathcal{T}$ and $\mathcal{U}$ range over trees; Tr denotes the class of all trees. The tree whose root is labelled I and whose subtrees are $\mathcal{T}_1, \ldots, \mathcal{T}_n$ is written $[\mathcal{T}_1, \ldots, \mathcal{T}_n]^I$.*

$$\frac{m = n}{a(b_1{:}S_1, \ldots, b_m{:}S_m).P|\overline{a}\langle c_1, \ldots, c_n\rangle.Q \to P[c_1, \ldots, c_n/b_1, \ldots, b_n]|Q} \quad (\textsc{Comm})$$

$$\frac{m \neq n}{a(b_1{:}S_1, \ldots, b_m{:}S_m).P|\overline{a}\langle c_1, \ldots, c_n\rangle.Q \to \mathit{wrong}} \quad (\textsc{Comm-Wrong})$$

$$\frac{P \to P'}{P|Q \to P'|Q} \quad (\textsc{Par})$$

$$\frac{P \to P'}{(\nu a{:}S)P \to (\nu a{:}S)P'} \quad (\textsc{Restr})$$

$$\frac{P \equiv_\pi P' \quad P' \to Q' \quad Q \equiv_\pi Q'}{P \to Q} \quad (\textsc{Eqv})$$

Figure 2.6: Reduction rules in the typed polyadic π-calculus

Definition 10 (Tree) *To each closed type S we associate an I/O-tree called Tree(S); it is the unique tree satisfying the following equations:*

$$\begin{array}{l} \mathit{if}\ S = (S_1, \ldots, S_n)^I \mathit{then}\ \mathit{Tree}(S) = [\mathit{Tree}(S_1), \ldots, \mathit{Tree}(S_n)]^I; \\ \mathit{if}\ S = \mu.A.S' \mathit{then}\ \mathit{Tree}(S) = [\mathit{Tree}(S'[\mu.A.S'/A]). \end{array}$$

Definition 11 (Tree simulation) *A relation $\mathfrak{R} \subseteq Tr \times Tr$ is a tree simulation if $(\mathcal{S}, \mathcal{T}) \in \mathfrak{R}$ implies:*

$$\begin{array}{l} \mathit{if}\ \mathcal{T}{=}[\mathcal{T}_1, \ldots, \mathcal{T}_n]^b\ \mathit{then}\ \mathcal{S} = [S_1, \ldots, S_n]^b \wedge \forall_{1 \le i \le n},\ \mathit{both}\ (\mathcal{S}_i, \mathcal{T}_i) \in \mathfrak{R}) \wedge (\mathcal{T}_i, \mathcal{S}_i) \in \mathfrak{R}); \\ \mathit{if}\ \mathcal{T}{=}[\mathcal{T}_1, \ldots, \mathcal{T}_n]^r\ \mathit{then}\ \mathcal{S} = [S_1, \ldots, S_n]^I\ \mathit{with}\ I \le r \wedge \forall_{1 \le i \le n}, (\mathcal{S}_i, \mathcal{T}_i) \in \mathfrak{R}); \\ \mathit{if}\ \mathcal{T}{=}[\mathcal{T}_1, \ldots, \mathcal{T}_n]^w\ \mathit{then}\ \mathcal{S} = [S_1, \ldots, S_n]^I\ \mathit{with}\ I \le w \wedge \forall_{1 \le i \le n}, (\mathcal{T}_i, \mathcal{S}_i) \in \mathfrak{R}); \end{array}$$

Definition 12 (Tree subtype relation) *The tree subtype relation $\mathcal{S} \le_{tr} \mathcal{T}$ is*

$$\mathcal{S} \le_{tr} \mathcal{T}\ \mathit{iff}\ (\mathcal{S}, \mathcal{T}) \in \mathfrak{R}\ \mathit{for\ some\ tree\ simulation}\ \mathfrak{R}.$$

Definition 13 (Subtype relation) *The subtype relation $\vdash S \le T$ is*

$$\vdash S \le T\ \mathit{if}\ \mathit{Tree}(S) \le_{tr} \mathit{Tree}(T).$$

The definition of the subtype relation is semantically natural and allows straightforward verification of some important properties, for example, transitivity. We write $\vdash S \gtrless T$ when $\vdash S \le T$ and $\vdash T \le S$. Pierce and Sangiorgi also define a finitary characterization of subtyping in terms of a deterministic algorithm. Σ ranges over subtyping contexts:

$$\Sigma ::= \emptyset \quad | \quad \Sigma, S \le t$$

Definition 14 (Algorithmic subtyping relation) *We define the algorithmic subtyping relation $\Sigma \vdash S \le T$ as the least relation closed under the rules in Figure 2.7 on the next page.*

These rules can be read as an algorithm by imposing an ordering on the rules. It attempts to construct a derivation of a subtyping judgement $\Sigma \vdash S \le T$ by applying the rules backwards in a goal directed fashion, using (Rec-L) and (Rec-R) in either order when both apply, but preferring (Ass) over (Rec-L) and (Rec-R) whenever both apply.

$$\frac{\forall_i \Sigma \vdash S_i \gtrless T_i}{\Sigma \vdash (S_1,\ldots,S_n)^b \leq (T_1,\ldots,T_n)} \quad \text{(BB)}$$

$$\frac{I \leq r \quad \forall_i \Sigma \vdash S_i \leq T_i}{\Sigma \vdash (S_1,\ldots,S_n)^I \leq (T_1,\ldots,T_n)^r} \quad \text{(XI)}$$

$$\frac{I \leq w \quad \forall_i \Sigma \vdash T_i \leq S_i}{\Sigma \vdash (S_1,\ldots,S_n)^I \leq (T_1,\ldots,T_n)^w} \quad \text{(XO)}$$

$$\Sigma, S \leq T \vdash S \leq T \quad \text{(Ass)}$$

$$\frac{\Sigma, \mu A.S \leq T \vdash S[\mu A.S/A] \leq T}{\Sigma \vdash \mu A.S \leq T} \quad \text{(Rec-L)}$$

$$\frac{\Sigma, S \leq \mu B.T \vdash S \leq T[\mu B.T//B]}{\Sigma \vdash S \leq \mu B.T} \quad \text{(Rec-R)}$$

Figure 2.7: Pierce and Sangiorgi's subtyping algorithm for the polyadic π-calculus

Properties of Subtyping

This concept of subtyping shows a close analogy between channel types and typed versions of λ-calculi [Gay93]. A function f with type $S \to T$ can be thought of a process that reads a value of type S from one location and writes a result type T to another. To start f, a pair of channels is sent along a request channel a_f, telling it where to find its argument and where to put the result. The type of a_f is $((S)^r, (T)^w)^w$. S occurs in a contravariant position and T in a covariant position. This typing validates the standard λ-calculus subtyping rule $S_1 \to T_1 \leq S_2 \to T_2$ if $S_2 \leq S_1$ and $T_1 \leq T_2$.

The subtyping relation is transitive. The subtyping algorithm is terminating and it can be shown that:

Theorem 1 (Completeness of the algorithm) *If Tree*$(S) \leq_{tr}$ *Tree*(T) *holds, then* $\vdash S \leq T$.

and

Theorem 2 (Soundness of the algorithm) *If* $\vdash S \leq T$ *then Tree*$(S) \leq_{tr}$ *Tree*(T).

These results shown by Pierce and Sangiorgi prove that subtyping by tree-simulation and the algorithmic subtyping relation are equivalent.

2.4.4 Typing

The typing rules are given in Figure 2.8 on the following page. A typing judgement $\Gamma \vdash P$:o asserts that "P is a well-behaved process under the assumptions Γ". If P is placed in an execution context whose free names support the protocol described by Γ, then the execution of P can never cause an error in communication.

The parallel composition of two processes is well behaved if each process is well-behaved in isolation. (The possibility of a bad communication between P and Q is detected as a failure by one or both to satisfy the requirements imposed by Γ (T-Par))

$$\Gamma \vdash \mathbf{0} : \circ \qquad \text{(T-NIL)}$$

$$\frac{\Gamma \vdash P : \circ \quad \Gamma \vdash Q : \circ}{\Gamma \vdash P|Q : \circ} \qquad \text{(T-PAR)}$$

$$\frac{\Gamma \vdash P : \circ}{\Gamma \vdash !P : \circ} \qquad \text{(T-REPL)}$$

$$\frac{\Gamma, a{:}S \vdash P : \circ}{\Gamma \vdash (\nu a{:}S)P : \circ} \qquad \text{(T-RESTR)}$$

$$\frac{\vdash \Gamma(a) \leq (S_1, \ldots, S_n)^r \quad \Gamma, b_1{:}S_1, \ldots, b_n{:}S_n \vdash P : \circ}{\Gamma \vdash a(b_1{:}S_1, \ldots, b_n{:}S_n).P : \circ} \qquad \text{(T-IN)}$$

$$\frac{\vdash \Gamma(a) \leq (\Gamma(b_1), \ldots \Gamma(b_n))^w \quad \Gamma \vdash P : \circ}{\Gamma \vdash \overline{a}\langle b_1, \ldots, b_n \rangle.P : \circ} \qquad \text{(T-OUT)}$$

Figure 2.8: Typing rules in the polyadic π-calculus

A replication of P is well behaved if a single copy is (T-REPL). A process whose outermost constructor is a restriction is well behaved if its body observes the constraints imposed by Γ and the declared typing of the new channel (T-RESTR).

The communication rules are more subtle: To ensure a well behaved input expression on a channel, it is necessary that the typing in the current context has the correct arity and the tuple of values read from the channel have correct types. This is expressed in the first condition in Rule (T-IN). The second condition checks if the channels behave consistently to the typings. The same is done for output expressions. To verify that an output expression is well behaved, the values that are sent must be allowed to be sent via the channel by using the subtyping relation.

2.5 Types for the π-Calculus Expressing Sequences of Messages

The translation of terms of the polyadic π-calculus to terms of the monadic π-calculus presented in Section 2.3.1 on page 12 is simple. The atomic communication of tuples of names is mapped to the communication of a sequence of names via a private channel. There is a need for a typing discipline as it is possible that components disagree about the length of tuples that shall be passed, for example $w(v).v(y,z)|\overline{w}\langle x\rangle.\overline{x}\langle a,b,c\rangle$. If P and Q are well typed polyadic processes, then the equivalence of $[\![P]\!]$ and $[\![Q]\!]$ implies the equivalences of P and Q, but the converse does not hold because there may be monadic contexts that do not conform to the interaction regime that underlies the translation. For example, $P{=}\overline{x}\langle y\rangle.\overline{x}\langle y\rangle$ and $Q{=}\overline{x}\langle y\rangle|\overline{x}\langle y\rangle$: The monadic processes are not barbed bisimilar in the monadic context $x(z).x(w).a(v)$.

The following two approaches present solutions for this problem. They allow to express sequences of names in the π-calculus.

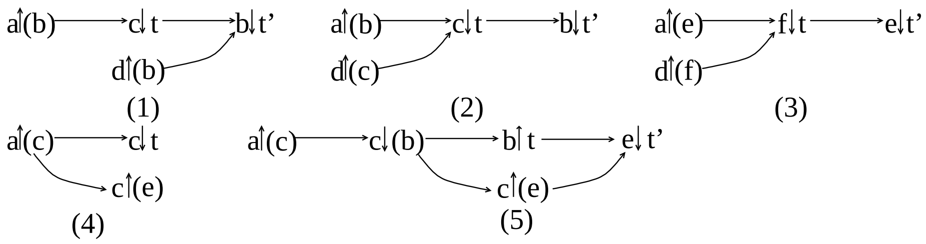


Figure 2.9: Examples for Graph types

2.5.1 Graph Types

Graph types are presented by Yoshida in [Yos96]. She uses graphs to represent the dynamic communication behavior of processes. Nodes denote atomic actions and edges denote the activation ordering between them, representing a deterministic protocol. Graph types try to capture dynamic behavior of name passing processes in simple graphical expressions.

Definition 15 (Graph) *Given a finite poset $\langle X, \leq \rangle$, its covering relation $\rightarrow$ is defined by: $x \rightarrow y$ if and only if $x \lneq y$ and there is no z with, $x \lneq z \lneq y$ where $x, y, z \in X$. Then $\langle X, \rightarrow \rangle$ gives a simple directed acyclic graph (equivalent to a Hasse diagram), from which $\leq$ is recovered as $\rightarrow^*$ (the reflexive and transitive closure) and $\lneq$ as $\rightarrow^+$ (the transitive closure). The term graph in the following section always means a graph presented by the covering relation. $x \rightarrow^* y$ says y is bigger than x. $G, G', \ldots$ denotes graphs, $N(G)$ is the set of nodes of G and $G_1 \uplus G_2$ is the (disjoint) graph union of G_1 and G_2.*

Definition 16 (Graph Types) *A graph type is a graph whose nodes are occurrences of the following atomic types:*

$$n ::= a{\downarrow}(b) \quad | \quad a{\uparrow}(b) \quad | \quad a{\downarrow}r \quad | \quad a{\uparrow}r$$

where $a, b, c, \ldots$ range over a set of names and $r, s, t, \ldots$ range over the set of base types. The self-inverse function $\overline{(\cdot)}$ is defined in the following way: $\overline{a \downarrow (b)} = a \uparrow (b)$ and $\overline{a \downarrow t} = a \uparrow t$. The function is extended to graph types so that the dual $\overline{G}$ is the result of replacing all occurrences of atomic types. Heads of a graph type are those occurrences of atomic types which are minimal with respect to the order $\rightarrow^+$.

Nodes in a graph type denote atomic actions of a process, while edges denote their synchronization ordering.

Examples

Simple graph types are $b \downarrow t \rightarrow a \uparrow t'$ and $b(c) \rightarrow a \uparrow t$. Other examples are given in Figure 2.9.

Definition 17 (Safety) *A graph type G is safe if, for any $t_1, t_2 \in N(G), fn(t_1) = fn(t_2)$ implies either $t_1 \rightarrow^+ t_2$ or $t_2 \rightarrow^+ t_1$*

Definition 18 (Pointed graph types) *A graph type G is pointed if: (1) there is a unique head in G, whose name is free in G; (2) all name occurrences except the above are bound, and (3) G is safe. $G_{\langle a \rangle}$ abbreviates that G is pointed with the name a free.*

Examples

In a safe graph type, two nodes with the same subject are strictly ordered so that, while the same subject may be used several times in interaction, at any time only one input and one output are active, so that interaction becomes deterministic. In Figure 2.9 on the page before, (4) is not safe because there is no ordering between $c \uparrow t$ and $c \uparrow (e)$.

Safe graph types can be shared if they are pointed. Pointedness says that an interaction starts at a single interaction point, and that, after the initial interaction, communications are done always using private channels. They are free from interference from a third party. An pointed graph is (5) in Figure 2.9 on the preceding page.

Definition 19 (Cut function) *$dom(\Psi)$ and $ran(\Psi)$ are the domain and the range of a partial function Ψ. Given G and G', assume $n, n_i \in N(G)$ and $n', n'_i \in N(G')$. Then a cut function Ψ from G to G' is a partial injection $\Psi : N(G) \rightharpoonup N(G')$ such that:*

- Duality*: If $n \in dom(\Psi)$, then $\Phi(n) = \overline{n}$.*
- Acyclicity*: $n_1 \rightarrow^+ n_2$ and $\Psi(n_2) \rightarrow^+ \Psi(n_1)$ cannot hold simultaneously for any $n_1, n_2 \in dom(\Psi)$.*
- Covering*: If $a \in fn(G) \cap fn(G')$, then $a \in fn(n)$ implies $n \in dom(\Phi)$. If there is any binding chain of a sequence $n_1 \succ\!\!\succ n_2 \cdots \succ\!\!\succ n_n$ in G and $n_i \in dom(\Phi)$ for some $i \leq n$, then for all $1 \leq j \leq n, n_j \in dom(\Phi)$, and then $\Phi(n_1) \succ\!\!\succ \cdots \succ\!\!\succ \Phi(n_n)$. Similarly for any binding chain in G' and Φ^{-1}.*

Operations on Graph Types

Prefix: Given a safe graph type G, $n \rightarrow G$ denotes the result of adding a new node n to G together with edges from the new node to all heads of G.

Parallel composition: Let safe graph types G_1, G_2 so that $G_1 \asymp G_2$ (denoting that there is a cut function from G_1 to G_2 when G_1 and G_2 are suitably α-converted) together with the corresponding cut function ψ be given. The *composition* $G_1 \odot G_2$ is a graph obtained by first identifying ψ-related nodes of $G_1 \uplus G_2$, getting a graph G' and then eliminating every identified node, preserving and reflecting $\rightarrow^*$ of G', deleting all redundant edges.

Figure 2.10 on the next page illustrates parallel composition of graph types. (1) and (2) are safe graph types. First, the same nodes are identified, in this case $c \downarrow t, c \uparrow t$ and $h \downarrow t, h \uparrow t$ (3). Then they are eliminated. The final graph is presented in (4).

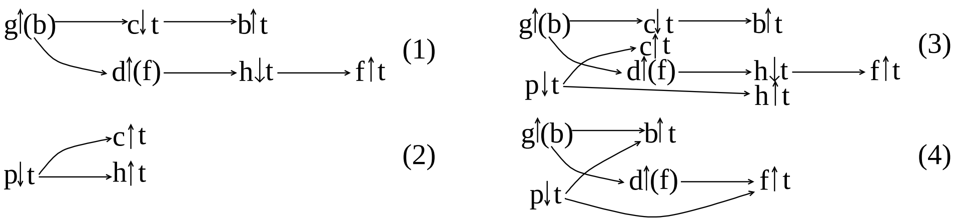


Figure 2.10: Cut elimination in Graph Types

Typing System

A *protocol type* $\alpha, \beta, \ldots$ is a set containing a pointed graph and its dual. The typing judgement $\Gamma \vdash P \triangleright_A G$ contains an environment Γ assigning free names in the π-term P to protocol types, A is a hidden name set and G is a graph type. The typing system is given in Figure 2.11 on the following page. The input and output rules consist of four pairs of rules. $a \downarrow (x) \to G$ represents a process that gets a new name x and acts with behavior G (in_1). In (out_1) x is recorded as a hidden name because it should be later restricted by the (res) rule. Rules (in_2) and (out_2) deal with a name a that gets or emits a name with a protocol type α and then acts like G. Rules (in_3) and (out_3) allow us to check graph types when a protocol type is successfully constructed. They are used for type inference, as are (in_4) and (out_4).

The key rule (par) allows us to extract forms of interaction structures of π-terms. Two graph types are combined using cut elimination. The names lost by cut elimination are kept to avoid further communication with these names. Even if the graphs G_1 and G_2 are sequential, the result may be a non-sequential graph. Rule (res) uses the hidden name set manipulated by ($\mathrm{out}_{1,3}$) and (par) and allows us to use names during type inference before the scope restriction occurs. The remaining rules are standard.

The following example term and graph type use the protocol type nat. We implement polyadic name passing of two names with $P \stackrel{\text{def}}{=} a(y).y(x_1).y(x_2).\overline{x_2}\langle x_1\rangle$. Then the corresponding types are $\vdash P \triangleright a \downarrow z \to z \downarrow nat \to z \downarrow (x_2) \to x_2 \uparrow nat$ or $\vdash P \triangleright a \downarrow (z) \to z \downarrow nat \to z \downarrow (nat)$.

The typing system π_G has the subject reduction property. Therefore, no type error can occur.

Fully abstract encoding for polyadic π-calculus

Yoshida proposes a fully abstract encoding translating also the types of the polyadic π-calculus. The mapping for terms is the same as in Section 2.3.1 on page 12, the mapping for types is:

$$\llbracket x \rrbracket \stackrel{\text{def}}{=} x,$$
$$\llbracket (S_1 \ldots S_n) \rrbracket \stackrel{\text{def}}{=} \{\downarrow (x) \to x \downarrow \llbracket S_1 \rrbracket \cdots \to x \downarrow \llbracket S_n \rrbracket, \uparrow (x) \to x \uparrow \llbracket S_1 \rrbracket \cdots \to x \uparrow \llbracket S_n \rrbracket\}$$

The mapping of terms and types is fully abstract. Yoshida proves the following theorem:

Theorem 3 $\Gamma \vdash P{=}Q \Leftrightarrow \llbracket \Gamma \rrbracket \vdash \llbracket P \rrbracket = \llbracket Q \rrbracket$

$$\frac{\Gamma \vdash P \rhd_A G}{\Gamma \vdash ax.P \rhd_A a{\downarrow}(x) \to G} \quad (1,2) \qquad (\textsc{In}_1)$$

$$\frac{x{:}\alpha, \Gamma \vdash P \rhd_A G}{\Gamma \vdash ax.P \rhd_A a{\downarrow}\alpha \to G} \quad (1) \qquad (\textsc{In}_2)$$

$$\frac{a{:}\beta, \Gamma \vdash P \rhd_A G}{a{:}\beta, \Gamma \vdash ax.P \rhd_A} \quad (2,3) \qquad (\textsc{In}_3)$$

$$\frac{a{:}(\alpha), x{:}\alpha, \Gamma \vdash P \rhd_A}{a{:}(\alpha), \Gamma \vdash ax.P \rhd_A} \qquad (\textsc{In}_4)$$

$$\frac{\Gamma \vdash P \rhd_A G}{\Gamma \vdash \overline{a}x.P \rhd_{A \cup \{x\}} a{\uparrow}(x) \to G} \quad (1,2) \qquad (\textsc{Out}_1)$$

$$\frac{x{:}\alpha, \Gamma \vdash P \rhd_A G}{x{:}\alpha, \Gamma \vdash \overline{a}x.P \rhd_A a{\uparrow}\alpha \to G} \quad (1) \qquad (\textsc{Out}_2)$$

$$\frac{a{:}\beta, \Gamma \vdash P \rhd_A \overline{G}}{a{:}\beta, \Gamma \vdash \overline{a}x.P \rhd_{A \cup \{x\}}} \quad (2,3) \qquad (\textsc{Out}_3)$$

$$\frac{a{:}(\alpha), x{:}\alpha, \Gamma \vdash P \rhd_A}{a{:}(\alpha), x{:}\alpha, \Gamma \vdash \overline{a}x.P \rhd_A} \qquad (\textsc{Out}_4)$$

$$\frac{\Gamma \vdash P \rhd_{A_1} G_1 \quad \Gamma \vdash P \rhd_{A_2} G_2}{\Gamma \vdash P|Q \rhd_{A_1 \cup A_2 \cup A} G_1 \odot G_2} \quad (4) \qquad (\textsc{Par})$$

$$\frac{\Gamma \vdash P}{\Gamma \vdash !P} \qquad (\textsc{Rep})$$

$$\vdash \mathbf{0} \qquad (\textsc{Nil})$$

$$\frac{\Gamma \vdash P \rhd_A G}{\Gamma \setminus a \vdash (a)P \rhd_{A \setminus \{a\}} G} \quad (5) \qquad (\textsc{Res})$$

$$\frac{\Gamma \vdash P \rhd_A G}{a{:}\alpha, \Gamma \vdash P \rhd_{A \cup \{b\}} G} \quad (6) \qquad (\textsc{Weak})$$

$$\frac{\Gamma \vdash P \rhd_A G \quad G \equiv_\alpha G' \Gamma \equiv_\alpha \Gamma'}{\Gamma' \vdash P \rhd_A G'} \qquad (\textsc{Alpha})$$

$$(1)\ a \notin \mathit{fn}(\Gamma) \cup A \quad (2)\ x \notin \mathit{fn}(\Gamma) \cup A \quad (3)\ \beta = \{{\downarrow}(x) {\to} G, {\uparrow}(x) {\to} \overline{G}\}$$
$$(4)\ \langle G_1, A_1 \rangle \asymp \langle G_2, A_2 \rangle,\ A = \mathit{fn}(G_1) \cap \mathit{fn}(G_2) \quad (5)\ a \notin \mathit{fn}(G)$$
$$(6)\ a \notin A \cup \mathit{fn}(G), b \notin \mathit{fn}(\Gamma) \cup \mathit{fn}(G)$$

Figure 2.11: Typing System for Graph Types π_G

2.5.2 Quaglia and Walker's Types for the Monadic π-Calculus

In the following monadic type system, a set of types $\mathcal{S}$ and a typing λ are fix.

Definition 20 (m-types) *The set of m-types, ranged over by $\sigma, \tau, \ldots$ is*

$$\mathcal{S}^m = \{\circ, \bullet\} \cup \{s^i \mid 1 \leq i \leq |\lambda(s)|, s \in \mathcal{S}\}.$$

For example, if $\mathcal{S} = \{s, t, r\}$ and $\lambda(s) = (t\ r)$, $\lambda(t) = (s)$ and $\lambda(r) = (r)$, then $\mathcal{S}^m = \{\circ, \bullet, s^1, s^2, t^1, r^1\}$.

Definition 21 (Graph) *The labelled directed graph $\mathcal{G}_\lambda$ has nodes $\mathcal{S}^m$, labels $\mathcal{S}$ and arrows*

$$\circ \xrightarrow{s} s^1 \xrightarrow{t_1} s^2 \xrightarrow{t_2} \cdots \xrightarrow{t_{n-1}} s^n \xrightarrow{t_n} \bullet$$

if $\lambda(s) = (t_1 \ldots t_n)$.

For the example typing above, the arrows are:

$$\circ \xrightarrow{s} s^1 \xrightarrow{t} s^2 \xrightarrow{r} \bullet \text{ and } \circ \xrightarrow{t} t^1 \xrightarrow{s} \bullet \text{ and } \circ \xrightarrow{r} r^1 \xrightarrow{r} \bullet.$$

Δ, Γ range over finite partial functions from names to $\mathcal{S}^m - \{\bullet\}$. Ψ ranges over finite partial functions from names to types. $n(\Psi)$ is the domain of Ψ. If $n(\Psi)=\{x_1, \ldots, x_n\}$ and $\Psi(x_i) = s_i$ for each i, $\{x_1{:}s_1, \ldots, x_n{:}s_n\}$ is written for Ψ. $\Psi(x) \simeq s$ means "if $x \in n(\Psi)$ then $\Psi(x)=s$". Ψ and Ψ' are *compatible* if $x \in n(\Psi) \cap n(\Psi')$ implies $\Psi(x)=\Psi'(x)$. If Ψ and Ψ' are compatible, Ψ, Ψ' stands for $\Psi \cup \Psi'$.

Definition 22 (Complementarity) *Suppose Δ_1 and Γ_1 are compatible, and Δ_2 and Γ_2 are compatible. Then Δ_1, Γ_1 and Δ_2, Γ_2 are complementary if $n(\Delta_1) \cap n(\Delta_2) = \emptyset, n(\Gamma_1) \cap n(\Gamma_2) = \emptyset$, and $(\Delta_1), (\Delta_2)$ and $(\Gamma_1), (\Gamma_2)$ are compatible.*

Definition 23 (λ^m-process) *A monadic process M is a λ^m-process if $\Psi; \Delta; \Gamma \vdash M$ may be inferred for some $\Psi; \Delta; \Gamma$ using the rules in Figure 2.12 on the following page.*

The rules for typing input and output are simpler than the rules for graph types. The typing of the input/output subject and object depend on the information in the environment. The rule for parallel composition requires that M_1 and M_2 have no communication with each other.

Like Yoshida, Quaglia and Walker are able to give a fully abstract encoding from the polyadic π-calculus to the monadic respecting types.

2.6 Discussion

The π-calculus is a powerful and well explored name passing calculus. The semantics of the calculus do not specify in which ordering messages are handled. Messages are sent via a channel. Another object may read the message from the channel any time. If there is no desired message, it waits. Therefore, the task of type systems in the π-calculus is to avoid wrong messages nobody wants to receive along this channel. If there would be the possibility that the message is correct, it may be sent, even if it is not received for an arbitrary time.

As there is no "built-in" type system, there are many proposals for type systems. Pierce and Sangiorgi's type system is based on the specification of the use of channels whether it is read only, write only or both. A subtyping relation is given, but the system is not able to express sequences of messages.

This problem is solved by Yoshida, whose graph types allow to express data dependencies via graphs. Different graph types can be combined to a new one which eliminates some dependencies with the cut function. However, all types are finite and deterministic.

$$\Psi;\emptyset;\emptyset \vdash \mathbf{0} \qquad \text{(NIL)}$$

$$\frac{\Psi;\emptyset;\{y{:}\sigma\} \vdash M}{\Psi, x{:}s;\emptyset;\{y{:}\circ\} \vdash \overline{x}\langle y\rangle.M} \quad \circ \xrightarrow{s} \sigma \wedge \Psi(x) \simeq s \wedge y \notin n(\Psi, x) \qquad (\text{OUT}_1)$$

$$\frac{\Psi;\emptyset;\{x{:}\tau\} \vdash M}{\Psi, y{:}s;\emptyset;\{x{:}\sigma\} \vdash \overline{x}\langle y\rangle.M} \quad \circ \neq \sigma \xrightarrow{s} \tau \wedge \Psi(y) \simeq s \wedge x \notin n(\Psi, y) \qquad (\text{OUT}_2)$$

$$\frac{\Psi;\{z{:}\sigma\};\emptyset \vdash M}{\Psi; x{:}s;\emptyset;\emptyset \vdash x(z).M} \quad \circ \xrightarrow{s} \sigma \wedge \Psi(x) \simeq s \wedge z \notin n(\Psi) \qquad (\text{INP}_1)$$

$$\frac{\Psi, z{:}s;\{x{:}\tau\};\emptyset \vdash M}{\Psi;\{x{:}\sigma\};\emptyset \vdash x(z).M} \quad \circ \neq \sigma \xrightarrow{s} \tau \wedge x, z \notin n(\Psi) \qquad (\text{INP}_2)$$

$$\frac{\Psi;\Delta_1;\Gamma_1 \vdash M_1 \quad \Psi;\Delta_2;\Gamma_2 \vdash M_2}{\Psi;\Delta_1,\Delta_2;\Gamma_1,\Gamma_2 \vdash M_1|M_2} \quad \Delta_1,\Gamma_1 \text{ and } \Delta_2,\Gamma_2 \text{ are complementary} \qquad (\text{PAR})$$

$$\frac{\Psi, z{:}s;\Delta;\Gamma \vdash M}{\Psi;\Delta;\Gamma \vdash (\nu z)M} \quad z \notin n(\Psi,\Delta,\Gamma) \qquad (\text{RES}_1)$$

$$\frac{\Psi;\Delta;\Gamma, z{:}\circ \vdash M}{\Psi;\Delta;\Gamma \vdash (\nu z)M} \quad z \notin n(\Psi,\Delta,\Gamma) \qquad (\text{RES}_2)$$

$$\frac{\Psi;\Delta, z{:}\sigma;\Gamma \vdash M}{\Psi;\Delta;\Gamma \vdash (\nu z)M} \quad z \notin n(\Psi,\Delta,\Gamma) \wedge \sigma \neq \circ \qquad (\text{RES}_3)$$

$$\frac{\Psi;\emptyset;\emptyset \vdash M}{\Psi;\emptyset;\emptyset \vdash !M} \qquad (\text{REP})$$

Figure 2.12: Monadic typing rules

Quaglia and Walker also present types that express sequences of messages, but their types are even more restrictive than graph types. They can combine two types only if they have no communication with each other.

Chapter 3

Types for Concurrent Objects

3.1 Introduction

Beside the π-calculus there are many other calculi for distributed systems: Some approaches try to extend sequential object-oriented calculi with primitives for synchronization. For example there are proposals for concurrent Eiffel [Mey93, Car93] and concurrent LOOM [Bro97]. A concurrent extension of Abadi and Cardelli's impς calculus [AC96b] is examined in Section 3.2.

A calculus based on channels like the π-calculus is TyCO [Vas99]. It supports asynchronous communication, abstractions and integrates a type system with type inference.

A different approach is Hewitt's Actor concept [HBS73, Agh86, AMST92]. It is based on actors which have a mail address and a message queue of received but yet not handled messages. This concept is different from the π-calculus as the uniqueness of the receiver of a message is guaranteed. (In the π-calculus, messages are sent along channels which may have multiple readers.) [GZ99a].

This property is used for some approaches of behavioral types. For example Colaço, Pantel and Sallé [CPS97] use a primitive actor calculus called CAP as a basis for their type system. Najm and Nimour's calculus of objects (see Section 3.6 on page 37) is also guaranteeing the unique receiver property. Process types which belong also to the class of behavioral types are also based on a calculus guaranteeing this property.

3.2 The concς Calculus

Abadi and Cardelli present in their book [AC96b] among other calculi the calculus impς for imperative objects. This calculus is based on sequential objects. Gordon and Hankin extend this calculus to concς for concurrent systems by adding parallel composition to impς. They sketch an encoding of this calculus to the π-calculus. The type system they use is also encodable in the type system of impς. They also propose to extend the calculus further with synchronisation primitives for mutual exclusion, but do not include them in the typing system.

$$
\begin{array}{rr}
(a \upharpoonright b) \upharpoonright c \equiv a \upharpoonright (b \upharpoonright c) & (\textsc{Assoc}) \\
(a \upharpoonright b) \upharpoonright c \equiv (b \upharpoonright a) \upharpoonright c & (\textsc{Proc-comm}) \\
(\nu p)(\nu q)a \equiv (\nu q)(\nu p)a & (\textsc{Switch}) \\
(\nu p)(a \upharpoonright b) \equiv a \upharpoonright (\nu p)b \quad \text{if } p \notin fn(a) & (\textsc{Extrude1}) \\
(\nu p)(a \upharpoonright b) \equiv ((\nu p)a) \upharpoonright b \quad \text{if } p \notin fn(b) & (\textsc{Extrude2}) \\
\text{let } x{=}(\text{let } y{=}a \text{ in } b)\text{in } c \equiv \text{let } y{=}a \text{ in(let } x{=}b \text{ in } c) \quad \text{if } y \notin fv(c) & (\textsc{Local}) \\
(\nu p)\text{let } x{=}a \text{ in } b \equiv \text{let } x{=}(\nu p)a \text{ in } b \quad \text{if } p \notin fn(c) & (\textsc{Restrict}) \\
a \upharpoonright \text{let } x{=}b \text{ in } c \equiv \text{let } x{=}(a \upharpoonright b) \text{ in } c & (\textsc{Let})
\end{array}
$$

Figure 3.1: Structural congruence in concς

3.2.1 Syntax and Operational Semantics of the concς-Calculus

Processes in concς consist of disjoint infinite sets of *names* $(p, q, \dots)$, *variables* $(x, y, \dots)$ and *labels* $(l, \dots)$. The sets of *results* (u, v), *denotations* (d) and *terms* (a, b, c) are defined by the following grammar:

$$
\begin{array}{l}
u ::= x \quad | \quad p \\
d ::= [l_i = \varsigma(x_i)b_i^{i \in 1..n}] \\
a ::= u \quad | \quad p \mapsto d \quad | \quad u.l \quad | \quad u.l \Leftarrow \varsigma(x)b \quad | \quad clone(u) \quad | \\
\qquad\quad let\ x = a\ in\ b \quad | \quad a \upharpoonright b \quad | \quad (\nu p)a
\end{array}
$$

The expression $p \mapsto [l_i = \varsigma(x_i)b_i^{i \in 1..n}]$ is a denomination of a process with name p. A method selection is denoted by $p.l$. A method update $p.l \Leftarrow \varsigma(x)b$ is an expression that updates the method labelled l in the object p. A clone $clone(p)$ is an expression that makes a shallow copy of p. In a method $\varsigma(x)b$, the variable x is bound. Its scope is b. In an expression $let\ x = a\ in\ b$, the variable x is bound, its scope is b. First a is run. If it returns a result, the result is called x and the b is executed. In $a \upharpoonright b$, the terms a and b run in parallel. In a restriction $(\nu p)a$, the name p is bound. Its scope is a.

The rules for structural congruence are given in Figure 3.1, the rules for reduction in Figure 3.2 on the following page. The management of parallel expressions that return results intends that the result of an expression is returned from the right hand side of the topmost parallel composition. Therefore, when a result is expected, parallel composition is not commutative. On the other side, in contexts immediately to the left of a parallel composition, where the result is ignored, parallel composition is commutative. This is expressed in Rule (Proc-comm).

Rule (Com) states that a denomination of a process is offering the methods all time, corresponding to a replicated selection in the π-calculus.

3.2.2 The Type System of concς

The type system distinguishes expressions (*Exp*) and processes (*Proc*). A process is simply a concurrent computation. An expression is a concurrent computation that is

$$
\begin{array}{rll}
(p{\mapsto}d)\upharpoonright p.l_j \rightarrow (p{\mapsto}d)\upharpoonright b_j[p/x_j] \quad \text{if } j{\in}1..n,\ (1) & & (\textsc{Com}) \\
(p{\mapsto}d)\upharpoonright (p.l_j{\Leftarrow}\varsigma(x)b) \rightarrow (p{\mapsto}d')\upharpoonright p \quad \text{if } j{\in}1..n, (1,2) & & (\textsc{Snd}) \\
(p \mapsto d)\upharpoonright clone(p) \rightarrow (p \mapsto d)\upharpoonright(\nu q)(q \mapsto d\upharpoonright q) \quad \text{if } q \notin fn(d)\ (1) & & (\textsc{Cln}) \\
\text{let } x{=}p \text{ in } b \rightarrow b[p/x] & & (\textsc{Let}) \\
(\nu p)a \rightarrow (\nu p)a' \quad \text{if } a{\rightarrow}a' & & (\textsc{Res}) \\
a\upharpoonright b \rightarrow a'\upharpoonright b \quad \text{if } a{\rightarrow}a' & & (\textsc{Par}) \\
\text{let } x{=}a \text{ in } b \rightarrow \text{let } x{=}a' \text{ in } b \quad \text{if } a{\rightarrow}a' & & (\textsc{Red}) \\
a \rightarrow b \quad \text{if } a \equiv a', a'{\rightarrow}b', b' \equiv b & & (\textsc{Str})
\end{array}
$$

$$(1)\ d = [l_i = \varsigma(x_i)b_i^{i\in 1..n}] \qquad (2) d'{=}[l_j{=}\varsigma(x)b, l_i{=}\varsigma(x_i)b_i^{i\in(1..n)-\{j\}}]$$

Figure 3.2: Reduction rules in concς

expected to return a result. A Type A can be either *Proc*, *Exp* or $[l_i{:}A_i^{i\in 1..n}]$, which is the type of objects with methods $l_1, \ldots, l_n$ that return results of type $A_1, \ldots, A_n$. The results of an expression may be ignored. Therefore, a term of type *Exp* is always a term of type *Proc*. The type system guarantees that a process does not occur in a context expecting an expression.

The subtype relation $A{<:}B$ is the least reflexive and transitive relation on types that satisfies $[l_i{:}B_i^{i\in 1..n+m}]{<:}[l_i{:}B_i^{i\in 1..n}]$, $[l_i{:}A_i^{i\in 1..n}]{<:}Exp$ and $Exp{<:}Proc$.

Definition 24 (Domain) *In concς, the domain $dom(a)$ of a is given by:*

$$
\begin{array}{ll}
dom(p \mapsto d) = \{p\} & dom(let\ x{=}a\ in\ b) = dom(a) \\
dom(a\upharpoonright b) = dom(a){\cup}dom(b) & dom((\nu p)a) = dom(a) - \{p\} \\
dom(a) = \emptyset \text{ for any other kind of } a &
\end{array}
$$

An *environment* E is a list $v_1{:}A_1, \ldots, v_n{:}A_n$; $E \vdash \circ$ means the results v_i are distinct. The typing judgement $E \vdash a{:}A$ is defined by the typing rules in Figure 3.3 on the next page.

3.3 The TyCO Calculus

3.3.1 Introduction

TyCO (Typed Concurrent Objects) is a calculus developed by Vasconcelos [Vas94b, VH93, VT93]. A general overview is given in [Vas99]. Programs are built from names and labels using asynchronous labelled messages, objects composed of collections of labelled methods, explicit concurrent composition and a restriction operator. There are also notions for process abstraction and variables over abstractions. They provide mutually recursive processes and allow us to declare an abstraction once and instantiate it several times.

The syntax is given in Section 3.3. The calculus is typed implicitly. A system for type inference is given in Section 3.4 on page 30. Types describe the kind of messages objects

$$\frac{E \vdash a{:}A \quad A{<:}B}{E \vdash a{:}B} \qquad \text{(Val Subsumption)}$$

$$\frac{E, u{:}A, E' \vdash \circ}{E, u{:}A, E' \vdash u{:}A} \qquad \text{(Val } u\text{)}$$

$$\frac{E \vdash U{:}[l_i{:}B_i^{i\in 1..n}] \quad j{\in}1..n}{E \vdash u.l_j{:}B_j} \qquad \text{(Val Select)}$$

$$\frac{E{=}E_1, p{:}[l_i{:}B_i^{i\in 1..n}], E_2 \quad E, x_i{:}[l_i{:}B_i^{i\in 1..n}] \vdash b_i{:}B_i \quad dom(b_i) = \emptyset \quad \forall i{\in}1..n}{E \vdash p \mapsto [l_i{=}\varsigma(x_i)b_i^{i\in 1..n}]{:}Proc}$$

$$\text{(Val Object)}$$

$$\frac{E \vdash u{:}A \quad E, x{:}[l_i{:}B_i^{i\in 1..n}] \vdash b{:}B_j \quad dom(b) = \emptyset \quad j{\in}1..n}{E \vdash u.l_j \Leftarrow \varsigma(x)b{:}A} \qquad \text{(Val Update)}$$

$$\frac{E \vdash u{:}[l_i{:}B_i^{i\in 1..n}]}{E \vdash clone(u){:}[l_i{:}B_i^{i\in 1..n}]} \qquad \text{(Val Clone)}$$

$$\frac{E \vdash a{:}A \quad E, x{:}A \vdash b{:}B \quad dom(b) = \emptyset \quad A{<:}Exp \quad B{<:}Exp}{E \vdash \text{let } x{=}a \text{ in } b{:}B} \qquad \text{(Val Let)}$$

$$\frac{E \vdash a{:}\mathrm{Proc} \quad E \vdash b{:}B}{E \vdash a \upharpoonright b{:}B} \quad dom(a){\cap}dom(b) = \emptyset \qquad \text{(Val Par)}$$

$$\frac{E, p{:}A \vdash a{:}B \quad p{\in}dom(a)}{E \vdash (\nu p)a{:}B} \qquad \text{(Val Res)}$$

Figure 3.3: Typing rules in concς

may receive, but not at what time a message may be received. A different proposal for types in TyCO from Ravara and Vasconcelos [RV97, RRV98] is presented in Section 3.5. We will continue examining the TyCO calculus beginning with Section 4.5 on page 56.

3.3.2 Syntax

TyCO expressions consist of a set of names $(a, b, c, \ldots, u, v, w, x, y, z)$, a set of abstraction variables $(X, Y, \ldots)$ and labels l. $P, Q, \ldots$ range over processes, letter A over process abstractions, and M over method collections. TyCO processes, methods and process abstractions are described using this grammar:

$$\begin{aligned} P &::= a!l{:}[\tilde{v}] \;\mid\; a?M \;\mid\; P|Q \;\mid\; \mathbf{0} \;\mid\; (\nu x)P \;\mid\; X[\tilde{v}] \;\mid\; A[\tilde{v}] \;\mid \\ &\quad\;\; \mathbf{def}\ X_1 = A_1\ \mathbf{and} \ldots \mathbf{and}\ X_n = A_n\ \mathbf{in}\ P \\ M &::= \{l_1{:}A_1 \& \ldots \& l_n{:}A_n\} \\ A &::= (\tilde{x})P \end{aligned}$$

where $n \geq 0$ and the labels $l_1, \ldots, l_n$ and also the abstraction variables $X_1, \ldots, X_n$ are pairwise distinct.

A process of the form $a!l{:}[\tilde{v}]$ is a *message*, name a is the target and $l{:}[\tilde{v}]$ is the communication of the message. The label l selects a particular method, where $\tilde{v}$ are the parameters. *Objects* are of the form $a?M$ where a is the *location* (name) of the object

$$P \equiv_T Q \quad (P\ \alpha\text{-convertible to } Q) \tag{Alpha}$$
$$P|Q \equiv_T Q|P \tag{Comm$_1$}$$
$$(l_1{:}A_1\&l_2{:}A_2) \equiv_T (l_2{:}A_2\&l_1{:}A_1) \tag{Comm$_2$}$$
$$(X_1 = A_1 \textbf{ and } X_2 = A_2) \equiv_T (X_2 = A_2 \textbf{ and } X_1 = A_1) \tag{Comm$_3$}$$
$$(P|Q)|R \equiv_T P|(Q|R) \tag{Assoc}$$
$$P|\mathbf{0} \equiv_T P \tag{Null$_1$}$$
$$(\nu x)\mathbf{0} \equiv_T \mathbf{0} \tag{Null$_2$}$$
$$\textbf{def } D \textbf{ in } \mathbf{0} \equiv_T \mathbf{0} \tag{Null$_3$}$$
$$(\nu xy)P \equiv_T (\nu yx)P \tag{Switch}$$
$$(\nu x)P|Q \equiv_T (\nu x)(P|Q) \quad (x \notin fn(Q)) \tag{Extr}$$
$$\textbf{def } D \textbf{ in } (\nu x)P \equiv_T (\nu x) \textbf{ def } D \textbf{ in } P \quad (x \notin fn(D)) \tag{Def$_1$}$$
$$(\textbf{def } D \textbf{ in } P)|Q \equiv_T \textbf{def } D \textbf{ in } P|Q \quad (fv(D) \cap fv(Q) = \emptyset) \tag{Def$_2$}$$
$$\textbf{def } D \textbf{ in def } D' \textbf{ in } P \equiv_T \textbf{def } D \textbf{ and } D' \textbf{ in } P \quad (fv(D) \cap fv(D') = \emptyset) \tag{Def$_3$}$$
$$\textbf{def } X = A \textbf{ and } D \textbf{ in } X[\tilde{v}]|P \equiv_T \textbf{def } X = A \textbf{ and } D \textbf{ in } A[\tilde{v}]|P \tag{Def$_4$}$$
$$((\tilde{x})P)[\tilde{v}] \equiv_T P[\tilde{v}/\tilde{x}] \tag{Abstr}$$

Figure 3.4: TyCO structural congruence rules

and M a collection of methods. $P|Q$ is the *parallel composition*, and $(\nu x)P$ is the *scope restriction*. $\mathbf{0}$ denotes the terminated process.

A *declaration* of the form **def** $X_1 = A_1$ **and** ... **and** $X_n = A_n$ **in** P binds abstractions A_i to variables X_i, defining processes and allowing to instantiate multiple copies of these processes in P. Abstractions of the form $(\tilde{x})P$ are not directly executable. An applied abstraction $A[\tilde{v}]$ denotes the process P with names in $\tilde{x}$ replaced by those in $\tilde{v}$. An applied abstraction variable $X[\tilde{v}]$ behaves as $A[\tilde{v}]$ if X is bound to A by some declaration.

Example

A one place buffer cell is an object that may hold a value and answer to two kinds of messages: write and read. The write message includes a value that shall be stored in the cell. The read message includes a name intended to receive the value the cell holds, simulating synchronous communication:

$$\begin{aligned}\textbf{def } \mathrm{Cell} = (\mathit{self},\ \mathit{value})\ \mathit{self}?\{&\mathrm{write}{:}(\mathit{newVal})\ \mathrm{Cell}[\mathit{self},\ \mathit{newVal}]\& \\ &\mathrm{read}{:}(\mathit{replyTo})\ \mathit{replyTo}!\mathit{val}{:}[\mathit{value}] \mid \mathrm{Cell}[\mathit{self},\ \mathit{value}]\}\end{aligned}$$

3.3.3 Operational Semantics

Definition 25 (Structural congruence) *Structural congruence $\equiv_T$ is the smallest congruence relation over processes generated by the rules in Figure 3.4.*

$$a!l_i{:}[\tilde{v}] | a?\{l_1{:}A_1\& \cdots \& l_1{:}A_1\} \to A_i[\tilde{v}] \quad (1 \le i \le n) \qquad (\textsc{Comm})$$

$$\frac{P \to P'}{P|Q \to P'|Q} \qquad (\textsc{Par})$$

$$\frac{P \to P'}{(\nu x)P \to (\nu x)P'} \qquad (\textsc{Res})$$

$$\frac{P \to P'}{\mathbf{def}\ D\ \mathbf{in}\ P \to \mathbf{def}\ D\ \mathbf{in}\ P'} \qquad (\textsc{Def})$$

$$\frac{P \equiv_T P' \quad P' \to Q' \quad Q' \equiv_T Q}{P \to Q} \qquad (\textsc{Str})$$

Figure 3.5: TyCO reduction rules

Structural congruence includes α-conversion, commutativity and associativity on processes, restrictions and definitions. Rules (3.12) and (3.13) together implement process instantiation. The first allows us to replace occurrences of X in P by its defining abstraction A; the latter substitutes the actuals by the formals. Process reduction does not constitute a reduction step.

Definition 26 (Reduction) *One-step reduction $\to$ is the smallest relation generated by the rules in Figure 3.5.*

Reduction $\twoheadrightarrow$ is the relation $\equiv_T \cup \to^+$, where $\to^+$ is the transitive closure of $\to$.

The rules are standard. Rule (Comm) describes the basic communication. The other rules describe restriction, parallel composition, structural congruence and definitions.

Replications in TyCO can be introduced in the following way:

$$\mathbf{def}\ X = a?\{l_1{:}(\tilde{x}_1)P_1|X\& \ldots \& l_n{:}(\tilde{x}_n)P_n|X\}\ \mathbf{in}\ X$$

where X does not occur free in $P_1, \ldots, P_n$. This replication corresponds to the replication rule of the π-calculus. This expression shall be abbreviated to:

$$a*?\{l_1{:}(\tilde{x}_1)P_1\& \ldots \& l_n{:}(\tilde{x}_n)P_n\}$$

3.4 Vasconcelos' Type-System for TyCO

3.4.1 Monomorphic Type Assignment

Types for names in TyCO are built from an infinite set of type-variables and the set of labels introduced by the calculus by means of a record and a recursive type constructor.

$t, t', \ldots \in \mathbb{V}$ range over type-variables, $\alpha, \beta, \ldots$ range over types, $\tilde{\alpha}, \tilde{\beta}, \ldots$ represent (finite, possibly empty) sequences of types. The set of types is given by the grammar

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

where $n \ge 0$ and $l_1, \ldots, l_n$ are pairwise distinct labels.

$$B, \Gamma_T \vdash v!l_i : [\tilde{v}] \quad (\Gamma_T(a) \approx \{l_1{:}\tilde{\alpha}_1, \ldots, l_n{:}\tilde{\alpha}_n\}, \Gamma_T(\tilde{v}) \approx \tilde{\alpha}_i) \tag{Msg}$$

$$\frac{B, \Gamma_T \vdash A_1 : \tilde{\alpha}_1 \ldots B, \Gamma_T \vdash A_n : \tilde{\alpha}_n}{B, \Gamma_T \vdash \{l_1 : A_1 \& \ldots \& l_n : A_n\} : \{l_1 : \tilde{\alpha}_1, \ldots, l_n : \tilde{\alpha}_n\}} \tag{Meth}$$

$$\frac{B, \Gamma_T \vdash M : \alpha}{B, \Gamma_T \vdash a?M} \quad (\Gamma_T(a) \approx \alpha) \tag{Obj}$$

$$\frac{B, \Gamma_T \cdot \tilde{x} : \tilde{\alpha} \vdash P}{B, \Gamma_T \vdash (\tilde{x})P : \tilde{\alpha}} \tag{Abs}$$

$$\frac{B, \Gamma_T \vdash P}{B, \Gamma_T \setminus x \vdash (\nu x)P} \tag{Res}$$

$$\frac{B, \Gamma_T \vdash P \quad B, \Gamma_T \vdash Q}{B, \Gamma_T \vdash P|Q} \tag{Par}$$

$$\frac{B, \Gamma_T \vdash A_1 : \tilde{\alpha}_1 \ldots B, \Gamma_T \vdash A_n : \tilde{\alpha}_n \quad B, \Gamma_T \vdash P}{B \setminus X, \Gamma_T \vdash \mathbf{def}\ X_1 = A_1\ \mathbf{and} \ldots \mathbf{and}\ X_n = A_n\ \mathbf{in}\ P} \quad (B(X_i) \approx \tilde{\alpha}_i) \tag{Def}$$

$$B \cdot X : \tilde{\alpha}, \Gamma_T \vdash X : \alpha \tag{Var}$$

$$\frac{B, \Gamma_T \vdash U : \tilde{\alpha}}{B, \Gamma_T \vdash U[\tilde{v}] : \tilde{\alpha}} \quad (\Gamma_T(\tilde{v}) \approx \tilde{\alpha}), U \text{ is an abstraction or abstraction variable} \tag{App}$$

Figure 3.6: TA: Monomorphic typing rules in TyCO

$[l_1 : \tilde{\alpha}_1, \ldots l_n : \tilde{\alpha}_n]$ describes a collection of names identifying objects containing methods labelled with $l_1, \ldots, l_n$. The arguments belong to the types in α_i.

Types are interpreted as rational trees. A type of the form $\mu.\alpha$ (with $\alpha \neq t$) denotes the rational tree solution of the equation $t = \alpha$. If α is a type, we denote by α^* its associated rational tree. An interpretation of recursive types as infinite trees induces an equivalence relation $\approx$ on types by putting $\alpha \approx \beta$ if $\alpha^* = \beta^*$.

Type assignments to names are formulas $x : \alpha$. *Typings* are denoted by Γ_T, Δ_T and are sets of type assignments where no name occurs twice. The typing $\Gamma_T \cup \{\tilde{x} : \tilde{\alpha}\}$ is abbreviated by $\Gamma_T \cdot \tilde{x} : \tilde{\alpha}$. Typing $\Gamma_T \setminus x$ is obtained by removing the type assignment for x from Γ_T. The expression $\Gamma_T(a)$ denotes the type α if $a : \alpha \in \Gamma_T$, and $\Gamma_T(a_1 \ldots a_n)$ represents the sequence of types $\alpha_1 \ldots \alpha_n$ if $a_i : \alpha_i \in \Gamma_T$, for $1 \leq i \leq n$.

Type assignments to abstraction variables are formulas $X{:}\tilde{\alpha}$. *Bases* B as sets of type assignments to variables where no variable occurs twice. Similarly to typings, $B \cdot X : \tilde{\alpha}$ is written for $B \cup \{X : \tilde{\alpha}\}$ if X does not appear in B. The notation $\Gamma_T \setminus X$ is written for the typing Γ_T with no occurrences of X and $B(X)$ is written for α if $X : \tilde{\alpha} \in B$.

Typing assignments to processes are statements $B, \Gamma_T \vdash P$, whereas *type assignment to abstractions* are statements $B, \Gamma_T \vdash A : \tilde{\alpha}$. The *monomorphic type system* TA is composed of the rules in Figure 3.6.

For example, the type of a cell abstraction described previously is

$$\text{Cell:}(\{\text{write:}t, \text{read:}\{\text{val:}t\}\}t), \emptyset \vdash A{:}\{\text{write:}t, \text{read:}\{\text{val:}t\}\}t$$

Cells are objects composed of two methods that hold values of type t. Method write accepts names of type t and method read accepts names capable of receiving val-labelled

$$\frac{B, \Gamma_T \vdash A : \sigma}{B, \Gamma_T \vdash A : \forall t.\sigma} \quad (t \text{ not free in } B, \Gamma_T) \qquad (\forall\text{-Intro})$$

$$\frac{B, \Gamma_T \vdash X : \forall t.\sigma}{B, \Gamma_T \vdash X : \sigma[\alpha/t]} \qquad (\forall\text{-Elim})$$

$$\frac{B, \Gamma_T \vdash A_1 : \sigma_1 \ldots B, \Gamma_T \vdash A_n : \sigma_n \quad B \setminus \tilde{X} \cdot \tilde{X} : \tilde{\sigma}, \Gamma_T \vdash P}{B \setminus \tilde{X}, \Gamma_T \vdash \ \mathbf{def}\ X_1 = A_1 \ \mathbf{and} \ldots \mathbf{and}\ X_n = A_n \ \mathbf{in}\ P} \quad (1) \qquad (\forall\text{-Def})$$

$$(1) B(X_i) \approx \tilde{\alpha}_i \text{ and } \sigma_i \text{ is of the form } \forall \tilde{t}_i.\tilde{\alpha}_i \, (1 \le i \le n)$$

Figure 3.7: $\mathrm{TA}_\forall$: Polymorphic typing rules in TyCO

messages carrying names of type t. The type variable t may be replaced by an arbitrary type α, or the type of the read method may be extended:

$$\text{Cell:}\{\text{write:}\alpha, \text{read:}\{\text{val:}\alpha, \text{foo:}\beta\}\}\alpha \vdash A\text{:}\{\text{write:}\alpha, \text{read:}\{\text{val:}\alpha, \text{foo:}\beta\}\}\alpha$$

3.4.2 Polymorphic Type Assignment

This assignment allows to abstract a type for an abstraction on a particular type-variable and to instantiate an abstracted type into different monomorphic types. Types for abstractions now fall into two classes: Monomorphic types as sequences of types for names and polymorphic types constructed using $\forall$. Polymorphic types $\sigma, \ldots$ are expressed by the following grammar:

$$\sigma \ ::= \ \tilde{\alpha} \quad | \quad \forall t.\sigma$$

Universal quantifiers can occur only at the top level of types. The types for abstractions are of the form $\forall t_1. \ldots .\forall t_n.\tilde{\alpha}$ for $n \ge 0$.

A type variable t occurs free in a type if t is not in the scope of a μt or a $\forall t$; otherwise it is bound. A type-variable may also occur free or bound in a typing or a basis.

The polymorphic type system $\mathrm{TA}_\forall$ is defined by the axioms and rules in system TA with monomorphic types in Rule (Var) (but not in Msg, Meth, Obj, Abs, App) replaced by the polymorphic types, by allowing polymorphic types in bases (but not in typings), by replacing Rule (Def) by the Rule ($\forall$-Def) below and by adding Rule ($\forall$-Intro) and Rule ($\forall$-Elim) in Figure 3.7. We may use Rule ($\forall$-Intro) to abstract the type of the value the cell holds and, therefore, obtain

$$\text{Cell:}(\{\text{write:}t, \text{read:}\{\text{val:}t\}\}t), \emptyset \vdash A\text{:}\{\text{write:}t, \text{read:}\{\text{val:}t\}\}t.$$

Then, by using Rule ($\forall$-Elim) twice, we get

$$\Gamma_T \vdash \mathbf{def}\ \text{Cell} = A \ \mathbf{in}\ \text{Cell}[c\,1] | \text{Cell}[c'\,\text{True}]$$

where Γ_T is the typing

$$\{c\text{:}\{\text{write:}Nat, \text{read:}\{\text{val:}Nat\}\}, 1\text{:}Nat, c'\text{:}\{\text{write:}Bool, \text{read:}\{\text{val:}Bool\}, \text{True:}Bool\}$$

for some types Nat and $Bool$.

Properties of the type system

The type system is implicit so that type information is inferred from the program.

Theorem 4 *(Subject reduction). If $\Gamma_T \vdash P$ and $P \twoheadrightarrow Q$, then $\Gamma_T \vdash Q$.*

Subject reduction ensures that a typing for a process does not change as the process is reduced.

Errors in TyCO occur when

$$P \twoheadrightarrow \textbf{def } D \textbf{ in } (\nu\tilde{y})(a?M|a!l:[\tilde{v}]|Q)$$

and l labels no method in M, or

$$P \twoheadrightarrow \textbf{def } D \textbf{ in } (\nu\tilde{y})(((\tilde{x})P)[\tilde{v}]|Q)$$

and the lengths of $\tilde{x}$ and $\tilde{v}$ do not match.

Corollary 5 *If $\Gamma_T \vdash P$, then P contains no messages that are not in the signature.*

The limitations of the type system can be shown with the following example which was taken from [CPS97]:

$$a{*}?\{\text{rep:}()\mathbf{0}\} \mid b{*}?\{\text{quest:}(x)x!\text{rep:}[]\} \mid c{*}?\{\text{rep:}()\mathbf{0}\ \&\ \text{msg:}()\mathbf{0}\} \mid b!\text{quest:}[a] \mid b!\text{quest:}[c]$$

The typing of this example fails because the two messages $b!$quest:$[a]$ and $b!$quest:$[c]$ contain arguments of different types being in subtyping relationship. This simple example can be typed when subsumption is supported.

A further example is the following one-place buffer:

$$\begin{aligned}\text{Empty}(b) &= b?\{\text{write:Full}(b,u)\}\\ \text{Full}(b,u) &= b?\{\text{read:}(r)r\triangleleft\text{val:}[u]|\text{Empty}(b)\}\end{aligned}$$

The type inference system gives b two different typings. This also results in an error. This example was the motivation for another version of types which are presented in the following section.

3.5 Ravara and Vasconcelos' Behavioral Types

The type system for behavioral types was developed by Ravara and Vasconcelos and presented in [RV97, RRV98]. They allow non-uniform types for TyCO processes. They use types that change their behavior like the one-place buffer given above. This is done by making the notation of error depend on the environment. In the following we do not take care of abstractions to keep the type system simple. They could be integrated easily following Section 3.4.1 on page 30.

3.5.1 Error-Processes

The notion of errors in processes is defined the following way:

Definition 27 (Context) *A context C is the concurrent composition of messages and a constant $[]$ (called the* hole*).*

Definition 28 (x-redex) *A process P has a x-redex if $P \equiv (\nu\tilde{u})(x*?M|x!m|Q)$. A process P has a bad x-redex if P has an x-redex and the label m is not defined.*

Definition 29 (Error) *A process P is an error, written $P \in \textsc{Err}$ if*

- *$\exists_C C(P) \twoheadrightarrow Q$ and Q has a bad x-redex for some x not in C, and*
- *$\forall_C C'[Q] \twoheadrightarrow R$ and R has a bad x-redex.*

Errors are processes with bad x-redexes that persist throughout reduction. An occasional bad x-redex is not enough to make the process an error. So it is possible that messages find their target. The calculus has *weak fairness* in the treatment of messages. Some examples shall be shown:

1. $S \stackrel{def}{=} \text{Empty}(x)|x!\text{read} : [u] \notin \textsc{Err}$ since, although the process S is a bad x-redex. We have $C[S] \rightarrow \text{Full}(xv)|\text{read} : [u]$ for $C \stackrel{def}{=} x!\text{write:}[v]|[]$ containing no bad x-redexes;

2. $P \stackrel{def}{=} y?\{b : x!a\}|x*?\{c\} \in \textsc{Err}$ since $C[P] \rightarrow x!a|x*?\{c\}$ with $C \stackrel{def}{=} y!b|[]$ and no context can undo the bad x-redex;

3. $Q \stackrel{def}{=} t?\{e : (y)y!a\}|w?\{l{:}u*?\{c\}\} \notin \textsc{Err}$ although $C[Q] \twoheadrightarrow R \stackrel{def}{=} u!a|u*?\{c\}$ for $C \stackrel{def}{=} z!e : [u]|[]w!l$ and R is a persistent bad redex; but note that u is in C, violating the condition in the definition, meaning that the environment is the generator of the error.

3.5.2 Types

A type is a graph whose nodes (states) can be interpreted as an object's interface and the arcs (transitions) as the invoked methods. The type of an object represents its possible life-cycles.

Definition 30 (Types) *The set $\mathcal{T}$ of types is inductively defined as follows:*

- *$\mathcal{D} \subseteq \mathcal{T}$ for $\mathcal{D}$ an initial algebra of some fixed data types;*
- *$(V, I, A) \subseteq \mathcal{T}$ where V is a nonempty set of nodes, $I \subseteq V$ is a nonempty set of initial nodes and $A \subseteq V \times (\mathcal{L} \times \mathcal{T}^*) \times V$ is a set of arcs labelled by $\mathcal{L} \times \mathcal{T}^*$. Graphs are directed and contain no isolated nodes. We further require that a graph with more than one initial node is the disjoint union of connected components, one for each initial node.*

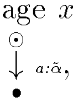
$\alpha^b, \beta^b, \gamma^b$ denote types. For a given graph α^b, the set V_{α^b} denotes its set of nodes, I_{α^b} denotes its set of initial nodes. A_{α^b} denotes its set of arcs. The label of an arc is denoted by t. Graphs are considered equal up to isomorphism on nodes.

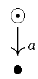
3.5.3 Typings

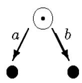
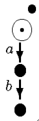
The *name-usage-triple* $x^* : \alpha^b$, with $* \in \{!, \downarrow, \uparrow\}$, is a formula denoting the assignment of type α^b to name x, location of an object ($\downarrow$), location of a replicated object ($!$), or destination of a message ($\uparrow$).

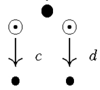
Definition 31 (Typing) *A typing Γ is a finite set of name-usage-type triples that has at most two occurrences of the same name, one as an object (replicated or not) and a second as a message.*

Examples

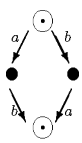
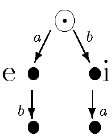
1. The message $x!a{:}\tilde{u}$ has a typing $\{x^{\uparrow}{:}(\{u_0, u_1\}, \{u_0\}, \{(u_0, a{:}\tilde{\alpha}, u_1)\}), \tilde{u}{:}\tilde{\alpha}\}$, graphically $\{x^{\uparrow}{:}$ , $\tilde{u}{:}\tilde{\alpha}\}$.

2. The object $x?\{a\}$ has a typing $\{x^{\downarrow}{:}$ $\}$. The input-name $x^{\downarrow}$ has the type

   $\alpha_1 =$  in $x?\{a\&b\}$ and $\alpha_2 =$  in $x?\{a{:}x{:}\{b\}\}$.

3. The input-name $x^{\downarrow}$ has the type  in $y?\{a{:}x?\{c\}\&b{:}x?\{d\}\}$; the graph is the union of two connected components, each one representing a possible behavior of the name x.

4. The parallel composition is dealt with in the following way: The input name $x^{\downarrow}$ has

   the type  in $x?\{a\}|x?\{b\}$, and the type  in $x?\{a{:}x?\{b\}\&b{:}x?\{a\}\}$.

   The first graph results from the interleaving of the graphs of the two objects. Intuitively each path represents a possible sequence of actions. The types are bisimilar.

If the parallel composition of two processes has a bad x-redex, then the processes should not be composed. Their typings are not compatible. But if the graphs of all output-names in a typing have some complete path that match with a projection of some path of the respective replicated input-name in the other typing, then there are no persistent bad x-redexes and the typings are compatible.

Definition 32 (Compatible typings) *Two typings Γ and Δ are compatible, denoted by $\Gamma \asymp \Delta$, if $\Gamma(x^!)$ has a projection equal to some complete path of $\Delta(x^\uparrow)$, and vice-versa.*

Definition 33 (Union) *The union $\Gamma \uplus \Delta$ of two compatible typings is the typing:*

1. $\Gamma \cup \Delta$, *if* $dom(\Gamma) \cap dom(\Delta) = \emptyset$;
2. $(\Gamma \cdot x^*{:}\gamma) \uplus (\Delta \setminus x^*)$, *if* $\Gamma(x^*) = \alpha, \Delta(x^*) = \beta$, *and* $\gamma \stackrel{def}{=} \alpha$ *if* $\alpha \sim \beta$ *and* $\alpha \uplus \beta$ *otherwise;*
3. $\{x^*{:}\alpha, x^\uparrow{:}\beta\} \cup ((\Gamma \setminus x^*) \uplus (\Delta \setminus x^\uparrow))$, *if* $\Gamma(x^*) = \alpha, \Delta(x^\uparrow) = \beta$, *and* $* \in \{!, \downarrow\}$;
4. $\{x^!{:}\alpha\} \cup ((\Gamma \setminus x^!) \uplus (\Delta \setminus x^\downarrow))$, *if* $\Gamma(x^!) = \alpha, \Delta(x^\downarrow) = \beta$, *and* $\alpha = repl(\gamma)$, *for some* γ *such that* $\gamma \sim \beta$;
5. $\{x^!{:}\alpha, x^\downarrow{:}\beta\} \cup ((\Gamma \setminus x^!) \uplus (\Delta \setminus x \downarrow))$, *if* $\Gamma(x^!) = \alpha$ *and* $\Delta(x^\downarrow) = \beta$).

Definition 34 (Interleaving) *The interleaving $\Gamma \| \Delta$ of two compatible typings is the typing:*

1. $\Gamma \cup \Delta$ *if* $dom(\Gamma) \cap dom(\Delta) = \emptyset$;
2. $(\Gamma \cdot x^*{:}\gamma) \| (\Delta \setminus x^*)$, *if* $\Gamma(x^*) = \alpha, \Delta(x^*) = \beta$ *and* $\gamma \stackrel{def}{=} \alpha \| \alpha$ *if* $\alpha \sim \beta$ *and* $\alpha \| \beta$ *otherwise,*
3. $\{x^*{:}\alpha, x^\uparrow{:}\beta\} \cup ((\Gamma \setminus x^*) \| (\Delta \setminus x^\uparrow))$, *if* $\Gamma(x^*) = \alpha, \Delta(x^\uparrow) = \beta$, *and* $* \in \{!, \downarrow\}$;
4. $\{x^!{:}\alpha\} \cup ((\Gamma \setminus x^!) \| (\Delta \setminus x^\downarrow))$, *if* $\Gamma(x^!) = \alpha, \Delta(x^\downarrow) = \beta$, *and* $\alpha = repl(\gamma)$, *for some* γ *such that* $\gamma \sim \beta$;
5. $\{x^!{:}\alpha, x^\downarrow{:}\beta\} \cup ((\Gamma \setminus x^!))$, *if* $\Gamma(x^!) = \alpha$ *and* $\Delta(x^\downarrow) = \beta$).

3.5.4 Typing Rules

Definition 35 (Behavioral type system) *The typing rules for the behavioral type system are given in Figure 3.8 on the next page, where $repl(\alpha)$ is the replication of a type (see [RV97]).*

A short description of the most important rules shall be given: The graph of a message has a single arc labelled with the message's label to the target. Arbitrary types are assigned to the carried names (Rule (Msg)). Rule (Obj) states that the graph of an object has as many arcs starting from the initial node as the number of initial nodes of its type in each of its methods. Each arc is labelled with the method's label. In the ending of each of those nodes we hang the graph of that object as it occurs in the corresponding method. The type of a replicated object is the greatest fix-point of the graph of the object (Rule (Rep)). The parallel composition of two processes (Rule (Par)) has a typing resulting from the interleaving of the component's typings. Rule (Weak) states that it is possible to add a type assignment formula with an output-name that does not occur in the typing to the typing of a process.

The typing rules guarantee the following properties:

$$\emptyset \vdash \mathbf{0} \qquad (\textsc{Nil})$$

$$\{\tilde{u}^* : \tilde{\alpha}, x^{\uparrow}:(\{v_0, v_1\}, \{v_0\}, \{(v_0, a : \tilde{\alpha}, v_1\})\} \vdash x \triangleleft a : [\tilde{u}] \quad (v_0 \neq v_1) \qquad (\textsc{Msg})$$

$$\frac{\Gamma_i \cdot \tilde{x}_i^{\uparrow}:\tilde{\alpha}_i \vdash P_i}{\biguplus_{1\leq i\leq n} \Gamma_i \uplus \{x^{\downarrow}:\alpha\} \vdash x?\{a_1:(\tilde{x}_1)P_1\& \ldots \&a_n:(\tilde{x}_n)P_n\}} \ (1) \qquad (\textsc{Obj})$$

$$\frac{\Gamma \cdot x^{\downarrow}:\alpha \vdash x?M}{\Gamma \cdot x^{!}:\ \mathit{repl}(\alpha) \vdash X*?M} \qquad (\textsc{Rep})$$

$$\frac{\Gamma \vdash P}{\Gamma \setminus x \vdash (\nu x)P} \qquad (\textsc{Res})$$

$$\frac{\Gamma \vdash P \quad \Delta \vdash Q}{\Gamma \| \Delta \vdash P|Q} \qquad (\textsc{Par})$$

$$\frac{\Gamma \vdash P}{\Gamma \cdot x^{\uparrow}:\alpha \vdash P} \qquad (\textsc{Weak})$$

(1)$(\asymp_{1\leq i\leq n} \Gamma_i) \asymp \{x^{\downarrow}:\alpha\}$ and α is such that $I_\alpha \stackrel{\text{def}}{=} \{u\}$ for a fresh node, and for $* \in \{!, \downarrow\}$

$$\begin{aligned}A_\alpha = &\cup_{1\leq i\leq n}\{(u, a_i:\tilde{\alpha}_i, w) | x^* \notin dom(\Gamma_i) \wedge w \text{ fresh}\}\cup\\ &\cup_1 \leq i \leq n\{(u, a_i:\tilde{\alpha}_i, v) | x^* \in dom(\Gamma_i) \wedge \forall v \in I_{\Gamma_i(x^*)}\}\cup\\ &\cup_{1\leq i\leq n} A_{\Gamma_i(x^*)}\end{aligned}$$

Figure 3.8: Behavioral type system

Theorem 6 (Subject reduction) *If* $\Gamma \vdash P$ *and* $P \twoheadrightarrow Q$*, then* $\exists_\Gamma \Gamma \vdash Q$ *and* $\Gamma \leq \Delta$ *(*$\leq$ *is a subtype relation based on subgraphs).*

Corollary 7 *If* P *is well-typed, then* $P \notin$ Err.

3.6 Behavioral Types of Najm and Nimour

The type system is based on an object calculus called OL [NN97, NN99, NNS99]. It is a variant of the π-calculus, but provides features such as interfaces, clients, servers and method invocation for the description of objects. It restricts the choice to input-prefix processes, enforces specialization of gates to input or output and tags method names with constant labels. The features of OL are also closely related to the Actors language.

3.6.1 The Object Calculus

The communication is asynchronous for public interfaces and rendez-vous for private interfaces. The formal syntax is given in Figure 3.9 on the following page.

The terminal T denotes an interface type explained later. The syntax of a service offer of an interface of an object is $u[m_1(\tilde{I}_1) = B_1, \ldots, m_n(\tilde{I}_n) = B_n]$. The object u waits for one of the messages $m_1, \ldots, m_n$ and then proceeds like B_j when the message m_j was sent. The invocation of such a service is written $u.m(\tilde{\rho}\tilde{v}) > B$. An interface creation is

$$
\begin{aligned}
I &::= u : T \\
Dcl &::= A[\tilde{I}] = B \\
Recep &::= u[m_1(\tilde{I}_1) = B_1, \ldots, m_n(\tilde{I}_n) = B_n] \\
B &::= \mathbf{0} \quad | \quad u.m(\tilde{\rho}\tilde{v}) > B \quad | \quad \textstyle\sum_{i=1}^{n} Recep_i \quad | \quad A[\tilde{\rho}\tilde{v}] \quad | \\
&\qquad \mathbf{create}\ A[\tilde{\rho}\tilde{v}] > B \quad | \quad \mathbf{new}\ u : T\ \mathbf{in}\ B \\
M &::= [u.m(\tilde{v})] \\
C &::= B \quad | \quad \mathbf{new}\ u : T\ \mathbf{in}\ C \quad | \quad M|C \quad | \quad C|C
\end{aligned}
$$

Figure 3.9: Object calculus Syntax

$$Recep_1 + Recep_2 \equiv_N Recep_2 + Recep_1 \qquad (\textsc{Com-Sum})$$

$$Recep_1 + (Recep_2 + Recep_3) \equiv_N (Recep_1 + Recep_2) + Recep_3 \qquad (\textsc{Assoc-Sum})$$

$$C_1|C_2 \equiv_N C_2|C_1 \qquad (\textsc{Com-Par})$$

$$C_1|(C_2|C_3) \equiv_N (C_1|C_2)|C_3 \qquad (\textsc{Assoc-Par})$$

$$C|\ \mathbf{0} \equiv_N C \qquad (\textsc{Null})$$

$$\mathbf{new}\ u_1{:}T_1\ \mathbf{in}\ (\mathbf{new}\ u_2{:}T_2\ \mathbf{in}\ C) \equiv_N \mathbf{new}\ u_2{:}T_2\ \mathbf{in}\ (\mathbf{new}\ u_1{:}T_1\mathbf{in}\ C)\ u \notin fn(C_2) \qquad (\textsc{New}_1)$$

$$\mathbf{new}\ u : T\ \mathbf{in}\ C \equiv_N \mathbf{new}\ v : T\ \mathbf{in}\ C[v/u]\ (v \notin fn(C)) \qquad (\textsc{New}_2)$$

Figure 3.10: Structural equivalence

given by **new** $u : T$ **in** B. The behavior of this expression is to create a new interface u of type T and then to behave like B. The behavior of objects can be named by using constructions of the kind $A[\tilde{I}] = B$. Objects that run in parallel with the creator object are created with **create** $A[\tilde{\rho}\tilde{v}] > B$.

Operational semantics

Interface name occurrences appearing in the scope of a behavior definition $A[\tilde{I}]{=}B$), a reception action $(u[m_1(\tilde{I}_1){=}B_1, \cdot\cdot, m_n(\tilde{I}_n){=}B_n])$ or an interface creation (**new** $u : T$ **in** C) are bound. Otherwise they are free.

The rules for the congruence relation $\equiv_N$ are given in Figure 3.10. Reductions may be annotated $C_1 \stackrel{u.m}{\to} C_2$ where u is an interface name and m a method name. Emitting actions on public interfaces will be noted "$_*$" whereas the emitting actions on private interfaces will remain noted ".".

The reduction rules are given in Figure 3.11 on the following page. Rules Rnd-Vous, Invoc and Msg describe rendez-vous communication on private interfaces and message passing on public interfaces. Rules Decl, Create and Conc describe instantiation, creation and the evolving of objects. Rule $\textsc{New}_1$ describes type updating on private interfaces. Rules $\textsc{New}_2$, $\textsc{New}_3$ and Struct state that the calculus has no side effects and may use $\equiv_N$ before and after a reduction.

$$u[\ldots, m(\tilde{v}{:}\tilde{T}) > B, \ldots] + \Sigma Recep_j | u.m(\tilde{w}) > B' \xrightarrow{u.m} B[\tilde{w}/\tilde{v}]B' \qquad (\textsc{Rnd-Vous})$$

$$u_{*}m(\tilde{v}) > B \longrightarrow B|[u.m(\tilde{v})] \qquad (\textsc{Invoc})$$

$$u[\ldots, m(\tilde{v}{:}\tilde{T})XS > B, \ldots] + \Sigma Recep_j|[u.m(\tilde{w})] \longrightarrow B[\tilde{w}/\tilde{v}] \qquad (\textsc{Msg})$$

$$\frac{A[\tilde{u}{:}\tilde{T}] \overset{Dcl}{=} B}{A[\tilde{v}] \longrightarrow B[\tilde{v}/\tilde{u}]} \qquad (\textsc{Decl})$$

$$\frac{A[\tilde{u}{:}\tilde{T}] \overset{Dcl}{=} B}{\textbf{create } A[\tilde{v}] > B' \longrightarrow B[\tilde{v}/\tilde{u}]|B'} \qquad (\textsc{Create})$$

$$\frac{C_1 \xrightarrow{a} C_1'}{C|C_1 \xrightarrow{a} C|C_1'} \qquad (\textsc{Conc})$$

$$\frac{C \xrightarrow{u.m} C' \quad T \xrightarrow{\rho.m(\tilde{T})} T'}{\textbf{new } u{:}T \textbf{ in } C \longrightarrow \textbf{ new } u{:}T' \textbf{ in } C'} \quad (\rho \in \{!, ?\}) \qquad (\textsc{New}_1)$$

$$\frac{C \xrightarrow{v.m} C'}{\textbf{new } u{:}T \textbf{ in } C \longrightarrow \textbf{ new } u{:}T \textbf{ in } C'} \quad (v \neq u) \qquad (\textsc{New}_2)$$

$$\frac{C \longrightarrow C'}{\textbf{new } u{:}T \textbf{ in } C \longrightarrow \textbf{ new } u{:}T \textbf{ in } C'} \qquad (\textsc{New}_3)$$

$$\frac{C_1' \xrightarrow{l} C_2' \quad C_1 \equiv_N C_1' \quad C_2 \equiv_N C_2'}{C_1 \xrightarrow{l} C_2} \qquad (\textsc{Struct})$$

Figure 3.11: Reduction rules

3.6.2 Behavior Types

An interface type is defined by a labelled transition system where each transition is a method signature. Interfaces can be public, providing uniform service availability or private, providing dynamically changing service offers. References to private interfaces must not be duplicated. Therefore, only one client at a time can use services of a private interface. Multiple clients must be coordinated to safely access private services.

An interface type specifies:

- the *behavior type* of the interface, i.e., the succession of the available services;
- the *mode* of the interface (public or private);
- the *role* of the interface (client or server). An object having the client role (noted !) of an interface can invoke services offered at this interface. An object having the server role (noted ?) can offer services on the interface.[1]

A behavior type is a triplet (E, x, r) noted $E \triangleright rx$, where x is the behavior type identifier, r is emitting/receiving capability ($r \subset \{!, ?\}$) and E a finite set of equations

[1]The client role of an interface can be called a reference to this interface, the server role can be called the interface.

of the form $x_k = e_k$ where each x_k is a behavior type identifier that appears once and only once in the left-hand side of an equation and e_k is an expression of the form:

$$e \ ::= \ \sum_{i=1}^{n} m_i(\tilde{\rho}_i \tilde{x}_i); x'_i \quad \text{with } l \neq l' \Rightarrow m_l \neq m'_l$$

where:

- each m_i is a method name. We call $\mathcal{M}$ the set of method names.
- each $\tilde{\rho}_i \tilde{x}_i$ is a list describing the behavior types of the arguments of method m_i. Each x_i is a behavior type identifier and ρ_i a role, i.e. client (!), server (?) or both (!?).
- each x'_i is a behavior type identifier.

Type behavior

A behavior type X that performs action a and then evolves to a new behavior type X' is denoted by the translation $X \xrightarrow{a} X'$. An action is a method signature annotated with a role: $\rho m(X_1, \ldots, X_n)$.

Behavior type transitions are defined by the following rule:

$$\frac{x = \sum_{i=1}^{n} m_i(\rho_i^1 x_i^1, \ldots, \rho_i^k x_i^k); x'_i}{r\, x \xrightarrow{\rho m_j(\rho_j^1 x_j^1, \ldots \rho_j^k x_j^k)} r\, x'_j} \quad (\rho \in \{!, ?\} \wedge \rho \subset r \wedge 1 \leq j \leq n)$$

The set of method names that can be handled by a behavior type X is generated by the function *possible*(X).

Definition 36 (Possible) *We define the set possible*$(X) = \{m \in \mathcal{M}, \exists X' \exists \tilde{Y} | X \xrightarrow{\rho m(\tilde{Y})} X'\}$.

Behavior Type Equivalence

The equivalence of behavior types is defined via bisimulation: Two behavior types are bisimilar if they can perform the same set of action types and for each action the resulting types are bisimilar.

Definition 37 (Bisimulation) *A binary relation β over behavior types is a bisimulation if $(X_1, X_2) \in \beta$ implies:*

- $X_1 \xrightarrow{\rho m(\tilde{Y}_1)} X'_1 \Rightarrow X_2 \xrightarrow{\rho m(\tilde{Y}_2)} X'_2$ *and* $(X'_1, X'_2) \in \beta, (\tilde{Y}_1, \tilde{Y}_2) \in \beta$
- $X_2 \xrightarrow{\rho m(\tilde{Y}_2)} X'_2 \Rightarrow X_1 \xrightarrow{\rho m(\tilde{Y}_1)} X'_1$ *and* $(X'_2, X'_1) \in \beta, (\tilde{Y}_2, \tilde{Y}_1) \in \beta$

Definition 38 (Behavior type equivalence) *Two behavior types X_1, X_2 are equivalent, noted $X_1 \sim_N X_2$, if and only if there exists a bisimulation β such that $(X_1, X_2) \in \beta$.*

Types that do not change their interface are called uniform:

Definition 39 (Uniform) *uniform*(X) *holds if* $X \xrightarrow{\rho m(\tilde{Y})} X'$ *implies* $X \sim_N X'$.

Behavior subtypes

Definition 40 (Simulation) *A binary relation* $\mathcal{S}$ *over behavior types is a simulation relation if* $(X_1, X_2) \in \mathcal{S}$ *implies:*

- $X_1 \xrightarrow{?m(\tilde{Y}_1)} X_1' \Rightarrow X_2 \xrightarrow{?m(\tilde{Y}_2)} X_2'$ *and* $(X_1', X_2') \in \mathcal{S}, (\tilde{Y}_1, \tilde{Y}_2) \in \mathcal{S}$
- $X_2 \xrightarrow{!m(\tilde{Y}_2)} X_2' \Rightarrow X_1 \xrightarrow{!m(\tilde{Y}_1)} X_1'$ *and* $(X_2', X_1') \in \mathcal{S}, (\tilde{Y}_2, \tilde{Y}_1) \in \mathcal{S}$

Informally, an interface type X_1 is a subtype of a type X_2 if and only:

client case: The processable messages of X_2 are a subset of the processable messages of X_1.

server case: The processable messages of X_1 are a superset of the processable messages of X_2.

Definition 41 (Subtypes) *A behavior type* X_1 *is a subtype of a behavior type* X_2*, noted* $X_1 \leq_N X_2$*, if there exists a simulation* $\mathcal{S}$ *such that* $(X_1, X_2) \in \mathcal{S}$.

3.7 Discussion

In this chapter, three quite different approaches for types in concurrent systems were presented. The first approach extends the calculus with return values impς of Abadi and Cardelli – constructed for sequential systems – with mechanisms for concurrency. The concept is based on passive objects, but it is possible to encode the π-calculus in concς. The authors did not give a fully abstract encoding due to problems with the observability of messages [GH98]. The type system guarantees that expected results of expressions are returned and every name points to an object.

TyCO is related to the π-calculus very closely. It can be shown that TyCO-terms can be encoded fully abstract to the π-calculus [VT93]. It provides abstractions that can be used to instantiate code more than one time. The type inference system is comfortable, but not very powerful. It specifies a set of messages that may be sent along a channel, but this set can never change.

The behavioral types of Ravara and Vasconcelos support changing sets of messages. But this is done for the sake of a relaxing error notation. An error because a message is sent occurs only if there is no context that may understand the message.

The third approach is a calculus created especially for behavioral types in distributed systems. The unique receiver property is an essential feature of this type system. It distinguishes between services accessible via a private interface (there may be at most one client having a reference to the interface) and services accessible via a public interface (an unlimited number may have references to the interface). The type of the public interface is uniform, the type of the private interface is non-uniform.

An arbitrary number of references to the public interface in this system may be passed because the services that may be accessed via this interface do not change. If the reference to a private interface is passed, the sender also passes the right to access this interface. Therefore, it is possible to keep in track with changes of this interface.

Part II

Process Types and the Process Type Calculus

Chapter 4

The Untyped Process Type Calculus

4.1 Introduction

Objects in a concurrent system communicate by passing sequences of messages. The set of acceptable messages may depend on previously received messages. Consider, for example, a buffer object understanding "put" and "get": If empty, the object accepts only a "put" message; "get" may lead to an error. We expect that a type system ensures statically that only acceptable messages are sent. This condition must hold even though in distributed systems it is possible that new objects are added at run time. Especially in concurrent systems it is important that the set of messages accepted by an object can change dynamically, expressing synchronization conditions [Nie93].

It is very difficult (if not impossible) to develop such type systems for process calculi like the π-calculus [MPW92]. For example, the type system developed by Pierce and Sangiorgi described in Section 2.4 on page 14 is quite powerful, but it cannot describe sequencing of values communicated along a channel. A possibility to express sequences of communication in the π-calculus are graph types presented in Section 2.5.1 on page 19. But graph types are just an observation of the behavior of the whole system and they are not able to express nondeterminism. The use of logical languages like the modal μ-calculus to express sequences [AD96] results in a complex type checking system which supports finite types only.

Types in the object-based process model of Vasconcelos and Honda [Vas94b, VH93] also cannot describe the sequencing of messages. The calculi proposed by Abadi and Cardelli [AC96b] focus on sequential object-oriented systems; hence, sequencing of messages is not even a topic for these calculi. A concurrent version of one of these calculi is presented in Section 3.2 on page 25, simply by extending an imperative object calculus with primitives for parallel composition and for synchronization via mutexes. This concurrent calculus even supports subtyping, but acceptable message sequences cannot be expressed in types.

On the other side, process types [Pun96a, Pun97a, Pun97b] support sequencing of messages for a calculus developed just for that purpose.

In the remainder of this chapter we will present the syntax of an untyped version of the process type calculus. The basic features of our calculus shall be presented and we will give a fully abstract encoding of our calculus to TyCO and vice versa. As the TyCO calculus can be encoded to the π-calculus, all we say for TyCO also holds for the π-calculus. A first version of the encoding which sketches the techniques was presented in [PP99a].

4.2 The Process Type Calculus

Our calculus is based on active objects behaving as in the Actor model [Hew77, Agh86, AMST92]. Every object has a unique name determined at the time of its creation. This name is used to specify the recipient of a message supporting object identity [KC86], a property of an object which distinguishes this object from all others. Object identity is a typical feature of object-oriented programming languages and is used as a basic dispatching mechanism in message passing.

Messages are sent asynchronously. That means, they travel a certain time until they reach the destination while the sending object accomplishes other tasks. The alternative is synchronous message passing where send and receive events are regarded as happening at the same time.

Concerning language design, it is easier to implement asynchronous message passing and the primitives of the underlying communication system offer more support. Languages for programming of distributed applications like the Actors model, or models based on the asynchronous π-calculus like Pict [PT96] and the join-calculus use asynchronous message passing [FG96].

It is quite easy to simulate the synchronous π-calculus in the asynchronous π-calculus: The sending process waits for an acknowledgement from the receiving process on a private channel. Basic results for an asynchronous version of the π-calculus were presented by Honda and Tokoro [HT91] and Boudol [Bou92].

Each object has a single identifier, a modifiable behavior and a single thread of execution. Different from the actor model, our communication system ensures that all messages are handled in the same (logical) ordering they were sent.

A simple scalar time representation ensuring this was proposed by Lamport in [Lam78]. His scalar time simply uses an incrementing clock to determine the ordering of messages. An overview of more complex mechanisms is given in [RS95].

A property of this communication mechanism is that when the receipt of a message changes the receiver's behavior (as specified in the receiver's type), the sender knows about the change already when sending the message. This property[1] is a precondition for process types [Pun97a].

Two infinite sets of names are considered given: $\mathbb{X}$ contains names (denoted by $a, b, c, \ldots$ and also $v, x, y \ldots$) used as object identifiers and parameters to be substituted by object identifiers. A set $\mathbb{C}$ contains labels $(l, \ldots)$ with $\mathbb{X} \cap \mathbb{C} = \emptyset$. Names are underlined if they occur at positions where object identifiers shall be substituted for them. An underlined occurrence of a name binds all following free occurrences of this name.

[1]Other communication mechanisms like synchronous message passing, rendezvous, procedure calls, etc. also have this property.

$$P ::= a\langle B/\tilde{\mu}\rangle \qquad \mu ::= l(\tilde{v}) \qquad M ::= l(\underline{\tilde{x}})B$$
$$B ::= \{\tilde{M}\} \quad | \quad a\triangleleft\mu.B \quad | \quad (\underline{a})@\mu.B \quad | \quad @\mu$$

Figure 4.1: Syntax of Objects

For each meta-symbol e, a sequence $e_1, \dots, e_n$ can be abbreviated by $\tilde{e}$ if n is arbitrary. Likewise, $\{\tilde{e}\}$ denotes the set containing $e_1, \dots, e_n$, and $|\tilde{e}|$ the length of the sequence. For each operator $\circ$, an expression $\tilde{e} \circ \tilde{g}$ (with $|\tilde{e}| = |\tilde{g}|$) stands for $e_1 \circ g_1, \dots, e_i \circ g_n$. All names in an underlined parameter list, for example $\underline{\tilde{x}}$, are pairwise different. The simultaneous substitution of e_i for all free occurrences of g_i (with $i = 1, \dots, |\tilde{e}|$) in f is denoted by $f[\tilde{e}/\tilde{g}]$, where $\tilde{e}$ and $\tilde{g}$ are sequences of the same length, and $\tilde{g}$ is a list of pairwise different names. The set of all names occurring free in any expression in $\{\tilde{e}\}$ is denoted by $\mathit{fn}(\tilde{e})$.

The syntax of objects is presented in Figure 4.1. An active object (or process, denoted by $P, Q, \dots$) of the form $a\langle B/\tilde{\mu}\rangle$ consists of an object identifier a, a current behavior B and a queue of received and not yet handled messages $\tilde{\mu}$.

A message $(\mu, \nu, \xi, \dots)$ of the form $l(\tilde{x})$ associates a message selector l with a (possibly empty) list of names $\tilde{x}$ used as actual parameters.

A method $(M, N, \dots)$ of the form $l(\underline{\tilde{x}})B$ specifies the behavior B preceded by a guard defining a method name (or message selector) l and formal parameters $\tilde{x}$. There are two kinds of methods: Globally visible creator methods are invoked when a new object is created or an object changes its own behavior. A message-handling method (that belongs to an object) is invokable if its guard matches the next message in the object's message queue. The invocation accepts a message sent to the object. The message used for invoking a creator message is specified in the behavior (following @). After invocation, the object behaves as B with the actual parameters substituted for the formal parameters.

A behavior (denoted by $B, C, \dots$) specifies if the object accepts a message and executes the corresponding message-handling method, sends a message, creates a new object or changes its behavior by invoking a creator method. An object behaving as $\{\tilde{M}\}$ selects a message-handling method in $\{\tilde{M}\}$ for invocation, depending on the next message in the queue. An object of behavior $a\triangleleft\mu.B$ sends a message μ to the object with identifier a and then behaves as B. One of behavior $(\underline{a})@\mu.B$ creates a new object (with a new object identifier substituting a) before behaving as B. The new object initially behaves as $@\mu$. The behavior $@\mu$ equals the behavior of a creator method matching μ after substituting actual parameters for formal parameters; this is, a creator method is invoked.

4.3 Reduction Semantics

Now we formalize the semantics of our object calculus. We define an execution relation on system configurations, these are sets of objects reflecting snapshots of object systems and a set of creator methods, representing a program.

A system configuration $\eta_{\{\tilde{M}\}}$ is a set of objects with pairwise different object identifiers. The creator method set $\{\tilde{M}\}$ in the subscript specifies the program executed by the object system the system configuration represents a snapshot of which.

Definition 42 (a-execution relation) *The a-execution relation $\eta_{\{\tilde{M}\}} \xrightarrow{a} \theta_{\{\tilde{M}\}}$ on system configurations is the least closure of these rules:*

$$\frac{\tilde{P} \equiv \tilde{P}' \quad \eta_{\{\tilde{M}\}} \cup \{\tilde{P}'\} \xrightarrow{a} \eta_{\{\tilde{M}\}} \cup \{\tilde{Q}'\} \quad \tilde{Q}' \equiv \tilde{Q}}{\eta_{\{\tilde{M}\}} \cup \{\tilde{P}\} \xrightarrow{a} \eta_{\{\tilde{M}\}} \cup \{\tilde{Q}\}} \quad (a\langle B/\tilde{\mu}\rangle \in \eta) \qquad (\text{Congr}_{\rightarrow})$$

$$\eta_{\{\tilde{M}\}} \cup @l(\tilde{x}) \xrightarrow{a} \eta_{\{\tilde{M}\}} \cup B \quad (l(\underline{\tilde{x}})B \in \{\tilde{M}\}) \qquad (\text{Call}_{\rightarrow})$$

$$\eta_{\{\tilde{M}\}} \cup a\langle \boldsymbol{\{} l(\underline{\tilde{x}})B, \tilde{N} \boldsymbol{\}} / l(\tilde{x}), \tilde{\mu}\rangle \xrightarrow{a} \eta_{\{\tilde{M}\}} \cup a\langle B/\tilde{\mu}\rangle \qquad (\text{Select}_{\rightarrow})$$

$$\eta_{\{\tilde{M}\}} \cup a\langle a \triangleleft \xi.B/\tilde{\mu}\rangle \xrightarrow{a} \eta_{\{\tilde{M}\}} \cup a\langle B/\tilde{\mu}, \xi\rangle \qquad (\text{Self send}_{\rightarrow})$$

$$\eta_{\{\tilde{M}\}} \cup \{a\langle b \triangleleft \xi.B/\tilde{\mu}\rangle, b\langle C/\tilde{\nu}\rangle\} \xrightarrow{a} \eta_{\{\tilde{M}\}} \cup \{a\langle B/\tilde{\mu}\rangle, b\langle C/\tilde{\nu}, \xi\rangle\} \qquad (\text{Send}_{\rightarrow})$$

$$\eta_{\{\tilde{M}\}} \cup \{a\langle(\underline{x})@\xi.B/\tilde{\mu}\rangle\} \xrightarrow{a} \eta_{\{\tilde{M}\}} \cup \{a\langle B[b/x]/\tilde{\mu}\rangle, b\langle @\xi[b/x]/\rangle\} \quad (b\langle C/\tilde{\mu'}\rangle \notin \eta) \qquad (\text{New}_{\rightarrow})$$

The execution relation $\eta_{\{\tilde{M}\}} \xrightarrow{*} \theta_{\{\tilde{M}\}}$ is the least reflexive and transitive closure of all a-execution relations $\eta_{\{\tilde{M}\}} \xrightarrow{a} \theta_{\{\tilde{M}\}}$ with $a \in \mathbb{X}$.

Rule (congr$_{\rightarrow}$) states that object congruence can be applied before and after each a-execution step if there is an object with name a in the environment η. The set $\{\tilde{M}\}$ is used by (call$_{\rightarrow}$) to replace invocations of creator methods with the corresponding behaviors. The message queue remains unchanged but the actual parameters $\tilde{y}$ are substituted for the formal parameters $\underline{\tilde{x}}$. The rules (select$_{\rightarrow}$) and (self send$_{\rightarrow}$) on objects accept messages and send messages to the objects themselves.

When an object sends a message to another object (send$_{\rightarrow}$), the message is appended to the receiver's message queue and the actual parameters $\tilde{y}$ are substituted for the formal parameters $\underline{\tilde{x}}$. A new object created using (new$_{\rightarrow}$) gets an empty message queue and a fresh constant name as object identifier. The name $\underline{x}$ is only a placeholder which will be substituted in the remaining behavior with the name b. Therefore, we guarantee the uniqueness of object identifiers but still allow techniques like scope extrusion.

Theorem 8 *Given a system configuration $\eta_{\{\tilde{M}\}}$ with pairwise different object identifiers, the execution of processes does not affect the pairwise difference of object identifiers.*

PROOF. The proof shall be given by induction. In the beginning the initial configuration $\eta_{\{\tilde{M}\}_0}$ contains only one object with an identifier a_0 and a method set $\{\tilde{M}\}$. The hypothesis is that a configuration containing $\eta_{\{\tilde{M}\}_n}$ contains only pairwise different object identifiers. Neither (call$_{\rightarrow}$), (select$_{\rightarrow}$), (self send$_{\rightarrow}$) nor (send$_{\rightarrow}$) rename object identifiers or create identifiers. The application of structural equivalence rules (congr$_{\rightarrow}$) may be applied only to already existing objects. Only the new operation creates a new identifier. This is done by renaming all further occurrences of the placeholder $\underline{x}$ in the later behavior of the object and the newly created object to b. Therefore, the identifier $\underline{x}$ may no longer be used to access any object. Although the identifier $\underline{x}$ was instantiated

$$\eta, M \cup \{l_1(\underline{\tilde{x}}).B\} \asymp \eta[l_2/l_1], M[l_2/l_1] \cup \{l_2(\underline{\tilde{y}}).C\} \quad (1) \qquad (\textsc{Subst})$$
$$\eta \cup \{x\langle B/\tilde{\mu}\rangle\}, M \asymp \eta \cup \{x\langle C/\tilde{\mu}'\rangle\}, M \quad (B \equiv C \wedge \tilde{\mu} \equiv \tilde{\mu}') \vee (B, \mu \ \alpha\text{-convertible to } C, \mu') \qquad (\textsc{Struct})$$
$$\eta \cup \{x\langle B/\rangle\}, M \asymp \eta \cup \{y\langle B/\rangle\}, M \quad (x, y \notin \{fn(\eta) \cup fn(B)\}) \qquad (\textsc{Send})$$
$$(1)\ (\underline{\tilde{x}} \equiv \underline{\tilde{y}} \wedge B[l_2/l_1] \equiv C \wedge l_1 \notin \{fn(\eta) \cup fn(M)\})$$

Figure 4.2: Configuration equivalence rules

more than one time, there cannot be any name conflicts because the identifier $\underline{x}$ is always replaced with a different fresh identifier. □

Definition 43 (Configuration equivalence) *The configuration equivalence, denoted by $\eta, M \asymp \theta, N$ is the smallest congruence under the rules in Figure 4.2.*

4.4 Encoding TyCO to the Untyped Process Type Calculus

A major difference between TyCO and the process type calculus is that correctly typed TyCO programs allow a location a to be read by more than one object (but these objects have to have the same type). In the process type calculus this is not allowed, because it would interfere with properties like type splitting and type updating (explained later). The solution is to simulate the multiple receiver property with placeholder objects in the process type calculus. A placeholder object receives all the messages (simulating a (channel) name in TyCO), and there may be an arbitrary number of objects requesting messages from the placeholder object. We use the same namespace as in TyCO. Therefore, we may use TyCO channel and method names also in the process type calculus.

This solution of the problem to translate an Actor-based system with unique receivers to a π-calculus-based system was also proposed by Gaspari and Zavattaro [GZ99a]. They translate their calculus based on actors to the asynchronous π-calculus using a not explicitly specified medium “ether” which does the same as our placeholder objects. They also state that their “ether” does not guarantee fairness.

The rules to create a configuration in the process type calculus from a TyCO program are given in Figure 4.3 on the following page and Figure 4.4 on page 51.

A TyCO program consists of objects (selections), restrictions, messages and abstractions which may contain again objects, restrictions, messages and abstractions. Abstractions correspond to creator methods in the process type calculus, while selections, restrictions and messages correspond to objects in the process type calculus.

We assume that the first step of the translation determines all the labels used in the TyCO program and stores them in the sequence $l_1, \ldots, l_n$. The rules creating process type objects are given in Figure 4.3 on the following page. They also insert creator methods in the configuration. The rules to create information for creator methods are given in Figure 4.4 on page 51.

$$\frac{\eta, M \vdash P \rightarrow \theta, N \quad \eta, M \vdash Q \rightarrow \theta', N'}{\eta, M \vdash P|Q \rightarrow \theta \cup \theta', N \cup N'} \tag{\textsc{Con}}$$

$$\eta, M \vdash a!l_i{:}\tilde{v} \rightarrow \eta \cup \{b\langle a \triangleleft l_i(\tilde{v}).\mathbf{\{\}}/\rangle\}, M \tag{\textsc{Snd}}$$

$$\frac{(M \vdash \triangle A_i \rightarrow B_i, \theta, N)_{1 \leq i \leq n}}{\eta \cup \{a\langle C'/\mu\rangle\}, M \vdash a?\{l_{i1}{:}A_1 \& \cdot\cdot \& l_{in}{:}A_n\} \rightarrow \eta \cup \{b\langle C/\rangle, a\langle C'/\mu, r(b)\rangle\}, M} \tag{\textsc{Sel}}$$

$$\frac{\eta \cup \{a\langle @\mathrm{rcv}/\rangle\}, M \vdash P[a/x] \rightarrow \theta, N}{\eta, M \vdash (\nu x)P \rightarrow \theta, N} \quad (a \notin \{fn(P) \cup fn(\eta) \cup fn(N)\}) \tag{\textsc{Res}}$$

$$\eta, M \vdash (\nu x)\mathbf{0} \rightarrow \eta, M \tag{\textsc{Res}\mathbf{0}}$$

$$\frac{(\eta, M_{i-1} \vdash \triangle A_i \rightarrow l_i(\tilde{x}_i)B_i, M_i \cup \{l_i(\tilde{x}_i)B_i\})_{1 \leq i \leq n} \quad \eta, M_n \vdash (Q[l_i/X_i])_{1 \leq i \leq n} \rightarrow \theta, N}{\eta, M_0 \vdash \mathbf{def}\ X_1 = A_1\ \mathbf{and} \ldots \mathbf{and}\ X_n = A_n\ \mathbf{in}\ Q \rightarrow \theta, N} \tag{\textsc{Def}}$$

$$\eta, M \vdash \mathbf{def}\ X_1 = A_1\ \mathbf{and} \ldots \mathbf{and}\ X_n = A_n\ \mathbf{in}\ \mathbf{0} \rightarrow \eta, M \tag{\textsc{Def}\mathbf{0}}$$

$$\eta, M \cup \{l(\underline{\tilde{y}})B\} \vdash (\tilde{y})l[\tilde{x}] \rightarrow \eta \cup \{a\langle @l[\tilde{x})/\tilde{y}]/\rangle\}, M \cup \{l(\underline{\tilde{y}})B\} \tag{\textsc{Var}}$$

$$\frac{M \vdash \triangle A \rightarrow l(\underline{\tilde{y}})B, N \cup \{l(\underline{\tilde{y}})B\}}{\eta, M \vdash A[\tilde{x}] \rightarrow \eta \cup \{a\langle B[x/y]/\rangle\}, N} \tag{\textsc{Abs}}$$

$$\eta, M \vdash \mathbf{0} \rightarrow \eta, M \tag{\textsc{Nul}}$$

$$C \equiv \mathbf{\{}l_{i1}(\underline{\tilde{x}}_{i1})B_1, .., l_{im}(\underline{\tilde{x}}_{im})B_n\mathbf{\}}$$

$$\mathrm{rcv}()\mathbf{\{}\mathrm{r}(\underline{y})@\mathrm{snd}(y), l_1(\underline{\tilde{x}}_1)\mathit{self} \triangleleft l_1(\tilde{x}_1).@\mathrm{rcv}(), .., l_n(\underline{\tilde{x}}_n)\mathit{self} \triangleleft l_n(\tilde{x}_n).@\mathrm{rcv}()\mathbf{\}}$$

$$\mathrm{snd}(\underline{y})\mathbf{\{}l_1(\underline{\tilde{x}}_1).y \triangleleft l_1(\tilde{x}_1).@\mathrm{rcv}(), .., l_n(\underline{\tilde{x}}_n).y \triangleleft l_n(\tilde{x}_n).@\mathrm{rcv}(), \mathrm{r}(\underline{y})\mathit{self} \triangleleft \mathrm{r}(y)@\mathrm{rcv}()\mathbf{\}}$$

Figure 4.3: Process construction rules

Process construction rules. The rules translate configurations of structure $\eta, M \vdash P \rightarrow \theta, N$. P is the TyCO program that shall be translated. η, M is a configuration in the process type calculus containing objects and creator methods. The resulting process type calculus configuration is θ, N.

At the begin of the translation, all messages $l_1, \ldots, l_n$ that may be sent while the program is executed have to be determined. They are used for the construction the creator methods rcv() and snd().

The syntax of TyCO supports an explicit parallel composition operator $P|Q$. The process type calculus does not provide the parallel composition operator. There is a set of objects in the environment executed concurrently. Therefore, we only have to join the two resulting configurations if we translate a TyCO parallel composition (Rule (Con)).

A TyCO process sending a message is translated to a process type object sending a message (Rule (Snd)). We are directly using TyCO names as names in our calculus for the receiver of the message and the actual parameters.

The rules (Res) and (Sel) can be used for the creation of new objects and the specification of their behavior. New objects are created the following way: A fresh name is created and we create a placeholder object for this name. This placeholder object (the behavior is given by the creator methods rcv() and snd()) receives all messages $(l_1, \ldots, l_n)$ that may be sent in the whole system. These messages are stored in a queue, by using the queue for incoming messages provided by the process type calculus. [2]

When a selection in TyCO is made on a channel, only the messages that are understood are considered. An arbitrary message of the acceptable messages is chosen. The messages not understood are postponed. In the process type calculus, the message r is sent to the corresponding placeholder object when a selection Rule (SEL) on a channel is used. Then the first message that is acceptable for the object (i.e., that is in $l_{i1}, \ldots, l_{im}$) in the queue of the placeholder object is sent to the object incorporating the selection. Messages that are not understood are inserted again at the end of the queue.

A placeholder can be accessed from more than one object. So we can simulate the multiple receiver property of TyCO channels. While every process type calculus object has a unique identifier, a channel may be shared by more than one object.

The selection instruction in TyCO associates a behavior to a name. We create an object (with a fresh name) which behaves as specified in the select instruction (the behavior is obtained by the abstract construction rules), and then tell the placeholder to send an appropriate message to the object. We do not preserve the ordering of messages because this is not required in TyCO programs (doing this would require a different placeholder object).

We have to handle restrictions on the empty process in a special rule in order to avoid placeholders for variables never used in Rule (RES**0**) and for empty definitions in Rule (DEF**0**).

Declarations of abstractions in TyCO are made using the **def**-statement. We use the abstract construction rules to get the behavior that is specified in the **def**-instruction and then insert a creator method with a fresh name and replace the TyCO name with the fresh name in the remaining TyCO program.

Abstractions can be used either by applications (Rule (ABS)) or via an applied abstraction variable (Rule (VAR)) when the variable was declared with a **def**-statement. In the latter case, there exists a creator method in the process type calculus environment that will be called. In the first case, the abstraction must be translated to obtain the behavior B. Then, an object with this behavior is created. The actual parameters are substituted for the formal parameters. The creator method which was inserted to the configuration by the abstract construction rules is removed from the configuration. Then, an object with that behavior is inserted in the environment.

Abstract construction rules. The abstract construction rules are used when the behavior of a declaration shall be obtained, but no object shall be created. They are of the form $M \vdash P \to B, N$, where M is a collection of creator methods in the process type calculus, P is a TyCO program, B is a process type calculus behavior and N is a set of creator methods. Abstract construction rules are called by putting a $\triangle$ before the TyCO program, and the calling process construction rules are Rule (DEF), Rule (SEL) and Rule (ABS).

Concurrent processes $P|Q$ in TyCO are translated to the creation of two processes behaving like the translations of P and Q. The names of the creator methods incorpo-

[2] The ordering of the messages is not guaranteed in the process type calculus because this is not required in TyCO.

$$\frac{M \vdash \triangle P \rightarrow B, N \quad M \vdash \triangle Q \rightarrow B', N'}{M \vdash \triangle P|Q \rightarrow (\underline{x})@l_1.(\underline{y})@l_2.\mathbf{\{\}}, N \cup N' \cup \{l_1()B, l_2()B'\}} \ (1)) \qquad (\triangle\ \textsc{Con})$$

$$M \vdash a!l_i{:}\tilde{v} \rightarrow a \triangleleft l_i(\tilde{v}), M \qquad (\triangle\ \textsc{Snd})$$

$$\frac{\forall j \in 1..n(M \vdash \triangle A_j \rightarrow B_j, N_j)}{M \vdash \triangle a?\{l_{i1}{:}A_1 \& \cdots \& l_{in}{:}A_n\} \rightarrow (\underline{b})@l().a \triangleleft \mathrm{r}(b).\mathbf{\{\}}, \bigcup_{1 \leq j \leq n} N_j \cup \{l()C'\}} \qquad (\triangle\ \textsc{Sel})$$

$$\frac{M \cup \{l()@\mathrm{rcv}()\} \vdash \triangle P[a/x] \rightarrow B, N}{M \vdash \triangle(\nu a)P \rightarrow (\underline{x})@l().B, N} \qquad (\triangle\ \textsc{Res})$$

$$M \vdash \triangle(\nu x)\mathbf{0} \rightarrow \mathbf{\{\}}, M \qquad (\triangle\ \textsc{Res}\mathbf{0})$$

$$\frac{(M_{i-1} \vdash \triangle A_i \rightarrow l_i(\tilde{x}_i)B_i, M_i \cup \{l_i(\tilde{x}_i)B_i\})_{1 \leq i \leq n} \quad M_n \vdash (\triangle Q[l_i/X_i])_{1 \leq i \leq n} \rightarrow B', N}{M_0 \vdash \ \mathbf{def}\ X_1 = A_1\ \mathbf{and} \ldots \mathbf{and}\ X_n = A_n\ \mathbf{in}\ Q \rightarrow B', N} \qquad (\triangle\ \textsc{Def})$$

$$M \vdash \triangle\ \mathbf{def}\ X_1 = A_1\ \mathbf{and} \ldots \mathbf{and}\ X_n = A_n \mathbf{in}\ \mathbf{0} \rightarrow \mathbf{\{\}}, M \qquad (\triangle\ \textsc{Def}\mathbf{0})$$

$$M \cup \{l(\tilde{x})B\} \vdash l[\tilde{x}] \rightarrow (\underline{y})@l(\tilde{x}).\mathbf{\{\}}, M \cup \{l(\tilde{x})B\} \qquad (\triangle\ \textsc{Var})$$

$$\frac{M \vdash \triangle A \rightarrow l(\underline{\tilde{y}})B, N \cup \{l(\underline{\tilde{y}})B\}}{M \vdash \triangle A[\tilde{x}] \rightarrow B[x/y], N} \qquad (\triangle\ \textsc{Abs})$$

$$\frac{M \vdash \triangle P \rightarrow B, N}{M \vdash \triangle(\underline{\tilde{x}})P \rightarrow l(\tilde{x})B, N \cup \{l(\underline{\tilde{x}})B\}} \quad (l \notin \{fn(N) \cup fn(B)\}) \qquad (\triangle\ \textsc{Apl})$$

$$M \vdash \triangle\mathbf{0} \rightarrow \mathbf{\{\}}, M \qquad (\triangle\ \textsc{Nul})$$

$$(1)\ (l_1, l_2 \notin \{fn(B) \cup fn(B') \cup fn(N) \cup fn(N')\})$$

Figure 4.4: Abstract construction rules

rating P and Q are fresh names (Rule ($\triangle$ CON)). A TyCO message is sent with the corresponding construct from the process type calculus, the names in the process type calculus are taken over from the names in TyCO (Rule ($\triangle$ SND)).

The selection and restriction work together in the same way as described in the previous paragraph. This time objects of that kind are created. The empty restriction (Rule ($\triangle$ RES**0**)) is mapped to an empty selection, as well as an empty definition (Rule ($\triangle$ DEF**0**)) or process (Rule ($\triangle$ NUL)).

A declaration of an abstraction is handled the same way as a declaration outside of a declaration, but Q is not executed directly (Rule ($\triangle$ DEF)).

Abstractions are handled the following way: An applied abstraction variable is a creation of a new object. An applied abstraction is translated using the abstract construction rules and then the behavior is returned (Rule ($\triangle$ VAR)). The constructed creator method is removed from the process type calculus configuration (Rule ($\triangle$ ABS)).

An application $\triangle(\tilde{x})P$ is translated and then a label and the parameters are added to the behavior B which corresponds to P. A creator method of that behavior is inserted to the process type calculus configuration (Rule ($\triangle$ APL)). The name must be selected carefully. As there is no equivalent name in TyCO, we have to ensure that this name is really unique.

Definition 44 (Expressive power) *Let Ψ_1, Ψ_2 be two process calculi, with respective equivalences $\approx_1 \subset \Psi_1 \times \Psi_1$ and $\approx_2 \subset \Psi_2 \times \Psi_2$. Ψ_2 is more expressive than Ψ_1 when there is a fully abstract encoding $\Psi_1 \to \Psi_2$: for all P, Q in Ψ_1, we have*

$$\begin{array}{c} P \to P' \\ Q \to Q' \\ P \approx_1 Q \Longleftrightarrow P' \approx_2 Q' \end{array}$$

Ψ_1 and Ψ_2 have the same expressive power when each one is more expressive than the other.

In the following we will show that the process type calculus has the same expressive power as the TyCO calculus using the structural equivalence given in Figure 3.4 on page 29 and the configuration equivalence given in Figure 4.2 on page 48.

Theorem 9 *For all processes P, Q in the TyCO-calculus and the equivalence relation $\equiv_T$, there is an encoding to the process type calculus where the resulting configurations θ_1, N_1 and θ_2, N_2 are equivalent with respect to $\asymp$, formally*

$$\begin{array}{c} \emptyset \vdash P \to \theta_1, N_1 \\ \emptyset \vdash Q \to \theta_2, N_2 \\ P \equiv_T Q \Longleftrightarrow \theta_1, N_1 \asymp \theta_2, N_2 \end{array}$$

PROOF. We first prove $P \equiv_T Q \implies \theta_1, N_1 \asymp \theta_2, N_2$. The TyCO equivalence relation shall be examined rule by rule if translations of programs equivalent by that relation are also equivalent. We consider twelve cases:

α-conversion: Is also supported by the process type calculus.

Commutative and associative parallel composition: The programs are translated to objects in our calculus. The rules concerning object commutativity and associativity hold in our calculus because we use sets of objects.

Transformations of the empty process: Processes of the kind **0** are not translated in the process type calculus configuration (NUL, $\triangle$ NUL). If there is a restriction on **0**, no placeholder is created (RES, $\triangle$ RES). A definition for a process **0** is not handled (DEF**0**, $\triangle$ DEF**0**).

Commutativity of scope restriction:

$$\begin{array}{ccc} P|(\nu xy)Q & \equiv_T & P|(\nu yx)Q \\ \downarrow & & \downarrow \\ \eta \cup \{a\langle B_1/\rangle, b\langle B_2/\rangle\}, M & \asymp & \eta' \cup \{b'\langle B_1'/\rangle, a'\langle B_2'/\rangle\}, M'\} \end{array}$$

We assume that objects with a behavior are created to the corresponding names. The a and b are the object identifiers on the right hand side and correspond to a' and b' on the left hand side. The behavior B_1 corresponds to B_1' on the left hand side, as well as B_2 corresponds to B_2'.

Extension of scope restriction:

$$\begin{array}{ccc} P|(\nu x)Q|R & \equiv_T & P|(\nu x)(Q|R) \quad (x \notin fn(R)) \\ \downarrow & & \downarrow \\ \eta \cup \{a\langle B/\rangle, M & \asymp & \eta' \cup \{a'\langle B'/\rangle\}, M' \end{array}$$

a and a' are fresh names and the extension of the restriction is preserved.

Extension of scope restriction in a definition:

$$\begin{array}{ccc} P|\ \mathbf{def}\ D\ \mathbf{in}\ (\nu x)Q & \equiv_T & P|\,(\nu x)\,\mathbf{def}\ D\ \mathbf{in}\ Q \quad (x\notin fn(D)) \\ \downarrow & & \downarrow \\ \eta\cup\{a\langle @\mathrm{rcv}()/\rangle, b\langle C'/\rangle\}, M\cup\{l_i(\underline{\tilde{x}}_i)B_i\} & \asymp & \eta'\cup\{a'\langle @\mathrm{rcv}()/\rangle, b'\langle C'/\rangle\}, M'\cup\{l_i(\underline{\tilde{x}}'_i)B'_i\} \\ (1\leq i\leq n) & & \end{array}$$

On the left hand side, the configuration contains a placeholder object a for x and an object b behaving like x (the behavior of b is specified in Q). The definitions of D are translated to the n creator methods l_i. A configuration equivalent up to α-conversion is created on the right hand side.

Extension of a definition: TyCO Rule (Def$_3$ on page 29) obviously holds in the process type calculus because the creator methods representing D are globally visibly, name conflicts in TyCO are solved by renaming (Def, $\triangle$ Def), and in the process type calculus always fresh names are used by Rule ($\triangle$ Apl on page 51).

Inflating definitions:

$$\begin{array}{ccc} P|\ \mathbf{def}\ D\ \mathbf{in}\ \mathbf{def}\ \ D'\ \mathbf{in}\ Q & \equiv_T & P|\ \mathbf{def}\ D\ \mathbf{and}\ D'\ \mathbf{in}\ Q \quad (\mathrm{fv}(D)\cap\mathrm{fv}(Q)=\emptyset) \\ \downarrow & & \downarrow \\ \eta\cup\{a\langle B/\rangle\}, M\cup\{l_i(\underline{\tilde{x}}_i)B_i, l_j(\underline{\tilde{x}}_j)B_j\} & \asymp & \eta'\cup\{a'\langle B'/\rangle\}, M'\cup\{l'_i(\underline{\tilde{x}}'_i)B'_i, l'_j(\underline{\tilde{x}}'_j)B'_j\} \\ (1\leq i\leq m \leq j\leq n) & & \end{array}$$

The behaviors of the objects and creator methods on both sides correspond to each other. The l_i are created by the translation of D. The l_j are created by the translation of D', the l'_i and l'_j from the translation of D and D', respectively.

Definition application:

$$\begin{array}{ccc} P|\ \mathbf{def}\ X = A\ \mathbf{and}\ D\ \mathbf{in}\ X[\tilde{v}]|Q & \equiv_T & P|\ \mathbf{def}\ X = A\ \mathbf{and}\ D'\ \mathbf{in}\ A[\tilde{v}]|Q \\ \downarrow & & \downarrow \\ \eta\cup\{a\langle @l(\tilde{v})/\rangle\}, M\cup\{l(\underline{\tilde{x}})B\} & \asymp & \eta'\cup\{b\langle @l'(\tilde{v})/\rangle\}, M'\cup\{l'(\underline{\tilde{x}}')B'\} \end{array}$$

On the left hand side, the behavior B corresponds to A (The parameters are obtained by processing A with Rule ($\triangle$ Apl on page 51)) and an object a is created which behaves like B. On the right hand side, the same configuration is created, with the difference, that the Rule ($\triangle$ Apl on page 51) is called a second time in order to translate $A[\tilde{v}]$.

Instantiation:

$$\begin{array}{ccc} ((\tilde{x})P)[\tilde{v}] & \equiv_T & P[\tilde{v}/\tilde{x}] \\ \downarrow & & \downarrow \\ \eta \cup \{a\langle B[\tilde{v}/\tilde{x}]/\rangle\}, M & \asymp & \eta' \cup \{a'\langle B'[\tilde{v}/\tilde{x}]/\rangle\}, M' \end{array}$$

On the left hand side the abstract translation in Rule (ABS on page 49) calls Rule ($\triangle$ APL on page 51). These rules return the behavior B (in a creator method, which is removed by Rule (ABS on page 49)) in which the actual parameters are substituted for the formal parameters. On the right hand side, the result is equal.

Commutativity of methods: A collection of methods is translated to a process type method selection which may also be permuted.

Commutativity of definitions: The definitions are translated into creator methods in a set. Therefore the ordering does not matter.

Now we prove $\theta_1, N_1 \asymp \theta_2, N_2 \Longrightarrow P \equiv_T Q$. We have to consider the properties of the sets which contain creator methods and objects. We have to consider seven cases:

Creator method renaming:

$$\begin{array}{ccc} \eta, M \cup \{l_1(\underline{\tilde{x}}).B\} & \asymp & \eta[l_2/l_1], M[l_2/l_1] \cup \{l_2(\underline{\tilde{y}}).B'\} \\ \downarrow & & \downarrow \\ \mathbf{def}\ X = A\ \ \mathbf{in}\ Q & \equiv_T & \mathbf{def}\ X' = A'\ \ \mathbf{in}\ Q' \end{array}$$

where $(\underline{\tilde{x}} \equiv \underline{\tilde{y}} \wedge B[l_2/l_1] \equiv C \wedge l_1 \notin \{fn(\eta) \cup fn(M)\})$.

This equivalence is working together with rule ($\triangle$ APL). The rule gives a unique name to a creator method that is a translation of an abstraction. When for example, the same TyCO program is translated twice, the name of the creator method is different, although they represent the same abstraction.

A and A' represent the behaviors B and B', respectively. Q and Q' are constructed using η, M and are therefore the same up to α-conversion.

Creator method commutativity:

$$\begin{array}{ccc} \eta, M \cup \{\mu_1, \mu_2\} & \asymp & \eta, M \cup \{\mu_2, \mu_1\} \\ \downarrow & & \downarrow \\ \mathbf{def}\ X{=}A_1\ \mathbf{and}\ Y{=}A_2\ \mathbf{and}\ D\ \mathbf{in}\ P & \equiv_T & \mathbf{def}\ X'{=}A_2'\ \mathbf{and}\ Y'{=}A_1'\ \mathbf{and}\ D'\ \mathbf{in}\ P' \end{array}$$

This holds because the agent names are α-convertible and the behaviors corresponding, the commutativity of definitions is given in Rule (COMM_3 on page 29).

Creator method associativity: Also holds, because the **and** statement is also associative.

Structural equivalence:

$$\begin{tikzcd} \eta \cup \{x\langle B/\rangle\}, M \arrow[d] \arrow[r, phantom, "\asymp"] & \eta \cup \{x\langle B'/\rangle\}, M \arrow[d] & (B \equiv B') \\ \textbf{def } D \textbf{ in } (P|Q) \arrow[r, phantom, "\equiv_T"] & \textbf{def } D' \textbf{ in } (P'|Q') & \end{tikzcd}$$

The D, P and D', P' are α-convertible. We have to examine the structural equivalences for behaviors B and B'. The sequence of the statements must remain unchanged, α-conversion for names and labels is possible. The ordering of methods in method selection in the process type calculus is commutative as in TyCO. Therefore, the transformation of every behavior B' that is structural equivalent to B in the process type calculus is also structural equivalent to a transformation of B in TyCO.

Object commutativity

$$\begin{tikzcd} \eta \cup \{x\langle B_1/\rangle, y\langle B_2/\rangle\}, M \arrow[d] \arrow[r, phantom, "\asymp"] & \eta \cup \{y\langle B_2/\rangle, x\langle B_1/\rangle\}, M \arrow[d] \\ \textbf{def } D \textbf{ in } (P|Q) \arrow[r, phantom, "\equiv_T"] & \textbf{def } D' \textbf{ in } (Q'|P') \end{tikzcd}$$

The D, P, Q are structural equivalent to D', P', Q', respectively. The parallel composition of processes is commutative in TyCO (Rule ($\textsc{Comm}_1$ on page 29)).

Object associativity

$$\begin{tikzcd} \eta\cup\{x\langle B_1/\rangle, (y\langle B_2/\rangle, z\langle B_3/\rangle)\}, M \arrow[d] \arrow[r, phantom, "\asymp"] & \eta\cup\{(x\langle B_1'/\rangle, y\langle B_2'/\rangle), z\langle B_3'/\rangle\}, M \arrow[d] \\ \textbf{def } D \textbf{ in } (P|(Q|R)) \arrow[r, phantom, "\equiv_T"] & \textbf{def } D' \textbf{ in } ((P'|Q')|R') \end{tikzcd}$$

The D, P, Q, R are structural equivalent to D', P', Q', R', respectively. The parallel composition of processes is associative in TyCO (Rule ($\textsc{Assoc}$ on page 29)).

Sending objects

$$\begin{tikzcd} \eta \cup \{x\langle B/\rangle\}, M \arrow[d] \arrow[r, phantom, "\asymp"] & \eta \cup \{y\langle B'/\rangle\}, M \quad (x, y \notin \{fn(\eta) \cup fn(B)\}) \arrow[d] \\ \textbf{def } D \textbf{ in } (P|Q) \arrow[r, phantom, "\equiv_T"] & \textbf{def } D' \textbf{ in } (P'|Q') \end{tikzcd}$$

The structural equivalence of D, P, Q and $D', P'Q'$ can be shown in the same way as in the case of structural equivalence. The difference is that the names of the process type calculus objects are not equal.

This means, that all the object does, is create new objects, send messages or change its behavior (but this behavior still receives no answers). Therefore, the name of the object is not important. $\square$

Theorem 10 *The algorithm given in Figure 4.3 on page 49 and Figure 4.4 on page 51 terminates.*

PROOF. Statements like send, select and instantiation of a variable are translated directly into the process type calculus. Restrictions, definitions and executions with empty processes are not translated to the process type calculus. Restrictions on non-empty processes are name substitutions. Definitions are translated to creator methods. While a normal statement results in an object performing the actions specified in the statement, statements in an agent definition are interpreted as statements in a creator method.

For every applied abstraction variable, the corresponding creator method is executed. An applied abstraction is translated abstractly and then executed. In the abstract translation, an applied abstraction variable is mapped to a new statement, an applied abstraction is translated abstractly and then called. Every TyCO expression is handled with one rule, the execution of the algorithm is deterministic and linear to the number of TyCO statements. □

4.5 Encoding the Untyped Process Type Calculus to TyCO

The rules to encode a configuration of the process type calculus are given in Figure 4.5 on the following page. A very important property is the message queue of every object which keeps the logical ordering of messages. TyCO does not provide such a queue. Therefore, the encoding of process types must provide the queue. The definitions used for the queue are made in $\mathcal{A}$. We use the first argument of the abstraction as a self-reference. Storage cells store a message, the message's parameters and the name of the next cell. To create an infinite number of cells, the abstraction Dq is used. This infinite number of cells is used for the creation of a queue structure. This is done by I. A queue Qq is containing a pointer to the next cell that shall be read and the next one that shall be written. The queue may receive put and get messages ("p" and "g") to store and retrieve cells.

The rules have the following structure: A (part of a process) type object is translated to a piece of TyCO-code. This code is either used in an abstraction or in a TyCO object. The process type object contains information about the object identity.

A process type calculus configuration contains a set of creator methods and a set of objects. The encoding starts with Rule (Creator). The creator methods are mapped to abstractions. The label names are translated to abstraction names. The formal arguments contain also a self-reference s_i.

Process type calculus objects are handled using Rule (Object). For every object a message queue is created using abstraction I. The name of the object is taken from the process type calculus. Then, the messages waiting for the object in its process type calculus queue are inserted in the queue. We use the acknowledgement message k to ensure that the ordering of the messages is kept. Then the TyCO object is executed concurrently, its behavior was obtained by using the remaining rules.

The identity of an object must be stored explicitly to be able to receive messages

$$
\begin{array}{c}
\mathcal{A} \stackrel{def}{=} \textbf{def } \mathrm{Cell} = (s{,}message{,}next)\ s?\{\mathrm{write}{:}(newmsg)\ \mathrm{Cell}[s{,}newmsg{,}next] \\
\&\mathrm{writenext}{:}(newnext)\ \mathrm{Cell}[s{,}message{,}newnext] \\
\&\mathrm{read}{:}(receiver{,}q)\ message!\mathrm{send} : [receiver] | q!\mathrm{shiftread}{:}[next] \\
\&\mathrm{readnext}{:}(q{,}passmsg)\ q!\mathrm{shiftwrite}{:}[next{,}passmsg] | \mathrm{Cell}[s{,}message{,}next]\} \\
\textbf{and } \mathrm{Qq} = (s{,}read{,}write)\ s?\{\mathrm{p}(message{,}replyto){:}\, write!\mathrm{readnext}[s{,}message] \\
| replyto!\mathrm{k}[] | \mathrm{Qq}[s{,}read{,}write] \\
\&\mathrm{g}(replyto) : \mathrm{read}!read[replyto{,}s] | \mathrm{Qq}[s{,}read{,}write] \\
\&\mathrm{shiftread}(newread) : \mathrm{Qq}[s{,}newread{,}write] \\
\&\mathrm{shiftwrite}(newwrite{,}info) : write!\mathrm{write}[info] | \mathrm{Qq}[s{,}read{,}newwrite]\} \\
\textbf{and } \mathrm{Dq} = (start)\ (\nu x) start!\mathrm{writenext}[x] | \mathrm{Dq}[x] \\
\textbf{and } \mathrm{I} = (name)(\nu sg)\mathrm{Dq}[s,g] | \mathrm{Qq}[name{,}g{,}g]
\end{array}
$$

$$
\frac{(s_i, B_i[X_1,..,X_n/l_1,..,l_n] \to P_i)_{1\le i\le n} \quad \eta[X_1,..,X_n/l_1,..,l_n] \to Q}{\eta, \{l_1(\underline{\tilde{x}}_1)B_1,..,l_n(\underline{\tilde{x}}_n)B_n\} \to \textbf{def } \mathcal{A} \textbf{ and } X_1{=}(s_1,\tilde{x}_1)P_1 \textbf{ and} \ldots\ X_n{=}(s_n,\tilde{x}_n)P_n \textbf{ in } Q} \quad (\textsc{Creator})
$$

$$
\frac{(x_i, B_i \to P_i)_{1\le i\le n}}{x_1\langle B_1/\tilde{\mu}_1\rangle,..,x_n\langle B_n/\tilde{\mu}_n\rangle \to \mathrm{I}[x_i] | ((\nu f)x_i!\mathrm{p}[a_{i1},f] | (f?\{\mathrm{k}{:}()(\nu f)x_i!\mathrm{p}[a_{i2},f]...\}) | P_i)_{1\le i\le n}} (1) \quad (\textsc{Object})
$$

$$
\frac{(x, B_i \to P_i)_{1\le i\le n}}{x, \mathbf{\{}l_1(\underline{\tilde{x}}_1)B_1,\ldots l_n(\underline{\tilde{x}}_n)B_n\mathbf{\}} \to (\nu a)x!\mathrm{g}(a) | a?\{l_1{:}(\tilde{x}_1)P_1\&\cdot\cdot\&l_n{:}(\tilde{x}_n)P_n\}} \quad (\textsc{Select})
$$

$$
\frac{x, B \to P}{x, a\triangleleft l(\tilde{x}).B \to (\nu sb)s?\{\mathrm{send}{:}(replyto)\ replyto!l(\tilde{x})\} | a!\mathrm{p}[s,b] | b?\{k() : P\})} \quad (\textsc{Send})
$$

$$
\frac{x, B \to P}{x, (\underline{a})@X(\underline{\tilde{x}}).B \to (\nu ag)(\mathrm{Dq}[g] | \mathrm{Qq}[a,g,g]) | P'[\tilde{x}/\tilde{y}] | P)} \quad (\textbf{def } X = (\tilde{y})P' \text{ defined}) \quad (\textsc{New})
$$

$$
x, @X(\underline{\tilde{y}}) \to P[x,\tilde{y}/s,\tilde{x}] \quad (\textbf{def } X = (s,\tilde{x})P \text{ defined}) \quad (\textsc{Become})
$$

$$
\begin{array}{c}
(1)\ a_{ij} \equiv_T a_{ij}?\{\mathrm{send}{:}(replyto) replyto!l_{ij}(x_{ij1},\ldots,x_{ijm'})\}, \\
\mu_i \equiv \mu_{i1},\ldots\mu_{im}, \mu_{ij} = l_{ij}(x_{ij1},\ldots,x_{ijm'})
\end{array}
$$

Figure 4.5: TyCO construction rules

(Rule (SELECT)). The queue for the corresponding object is called to get the next waiting message. A message is sent by adding the message to the queue object (Rule (SEND)).

A new object is created only if there is an abstraction corresponding to the called creator method. A new message queue is created, which is initially empty. The actual parameters replace the formal parameters (Rule (NEW)). A become-operation is executed by replacing the behavior of the object. The message queue is the queue from the old object. This is done by keeping the object identity as self-reference (Rule (BECOME)).

Theorem 11 *For all configurations η, M and θ, N in the untyped process type calculus and the equivalence relation $\asymp$, there is an encoding to TyCO where the resulting* *programs P and Q are equivalent with respect to $\equiv_T$, formally*

$$\begin{array}{c} \eta, M \rightarrow P \\ \theta, N \rightarrow Q \\ \eta, M \asymp \theta, N \Longleftrightarrow P \equiv_T Q \end{array}$$

PROOF. We first prove $\eta, M \asymp \theta, M \Longrightarrow P \equiv_T Q$. We have to consider the properties of the sets which contain creator methods and objects. We consider seven cases:

Creator method renaming:

$$\begin{array}{ccc} \eta, M \cup \{l'(\underline{\tilde{x}}).B\} & \asymp & \eta, M \cup \{l'(\underline{\tilde{y}}).B'\} \\ \downarrow & & \downarrow \\ \textbf{def } D \textbf{ and } X = A \textbf{ in } Q & \equiv_T & \textbf{def } D' \textbf{ and def } X' = A' \textbf{ in } Q' \end{array}$$

where $(\underline{\tilde{x}} \equiv \underline{\tilde{y}} \wedge B[l_2/l_1] \equiv C \wedge l_1 \notin \{fn(\eta) \cup fn(M)\})$.

D and D' are definitions made by translating μ and M. A and A' represent the behaviors B and B', respectively. Q and Q' are constructed using η and are therefore α-convertible.

Creator method commutativity:

$$\begin{array}{ccc} \eta, M \cup \{\mu_1, \mu_2\} & \asymp & \eta, M \cup \{\mu_2, \mu_1\} \\ \downarrow & & \downarrow \\ \textbf{def } D \textbf{ and } X{=}A_1 \textbf{ and } Y{=}A_2 \textbf{ in } P & \equiv_T & \textbf{def } D' \textbf{ and } X'{=}A_2' \textbf{ and } Y{=}A_1' \textbf{ in } P' \end{array}$$

This is true because the agent names are α-convertible and the behaviors corresponding, the commutativity of definitions is given in Rule ($\text{C}\textsc{omm}_3$ on page 29).

Creator method associativity: Also holds, because the **and** statement is also associative.

Structural equivalence:

$$\begin{array}{ccc} \eta \cup \{x\langle B/\tilde{\mu}\rangle\}, M & \asymp & \eta \cup \{x\langle B'/\tilde{\mu}'\rangle\}, M \\ \downarrow & & \downarrow \\ \textbf{def } D \textbf{ in } (P|Q|\mathrm{Qq}[x,r,w]) & \equiv_T & \textbf{def } D' \textbf{ in } (P'|Q'|\mathrm{Qq}[x,r',w']) \end{array}$$

The D, P and D', P' are α-convertible. We have to examine the possible structural equivalences for behaviors B and B'. The sequence of the statements must remain unchanged, α-conversion for names and labels is possible. The ordering of methods in method selection in the process type calculus is commutative as in TyCO. Therefore, the transformation of every behavior B' that is structural equivalent to B in the process type calculus is also structural equivalent to a transformation of B in TyCO. We also have to examine the message queues for x (the name x remains unchanged in both cases. The $\mathrm{Qq}[x,r,w]$ and $\mathrm{Qq}[x,r',w']$ are constructed following the Rule (Object).

Object commutativity

$$\begin{array}{ccc}
\eta \cup \{x\langle B_1/\tilde{\mu}\rangle, y\langle B_2/\tilde{\nu}\rangle\}, M & \asymp & \eta \cup \{y\langle B_2/\tilde{\nu}\rangle, x\langle B_1/\tilde{\mu}\rangle\}, M \\
\downarrow & & \downarrow \\
\textbf{def } D \textbf{ in} & & \textbf{def } D' \textbf{ in} \\
(P|\mathrm{Qq}[x, r_1, w_1]|Q|\mathrm{Qq}[y, r_2, w_2]) & \equiv_T & (Q'|\mathrm{Qq}[y, r'_2, w'_2]|P'|\mathrm{Qq}[x, r'_1, w'_1])
\end{array}$$

The D, P, Q are structural equivalent to D', P', Q', the same holds for the queues. The parallel composition of processes is commutative in TyCO (Rule ($\textsc{Comm}_1$ on page 29)).

Object associativity

$$\begin{array}{ccc}
\eta\cup\{x\langle B_1/\mu_1\rangle,(y\langle B_2/\mu_2\rangle, z\langle B_3/\mu_3\rangle)\},N & \asymp & \eta\cup\{(x\langle B'_1/\mu'_1\rangle, y\langle B'_2/\mu'_2\rangle), z\langle B'_3/\mu'_3\rangle\},N \\
\downarrow & & \downarrow \\
\textbf{def } D \textbf{ in } (P|\mathrm{Qq}[x, r_1, w_1]| & & \textbf{def } D' \textbf{ in } ((P'|\mathrm{Qq}[x, r'_1, w'_1]| \\
(Q|\mathrm{Qq}[y, r_2, w_2]|R\mathrm{Qq}[z, r_3, w_3])) & \equiv_T & Q'|\mathrm{Qq}[y, r'_2, w'_2])|R'|\mathrm{Qq}[z, r'_3, w'_3])
\end{array}$$

The D, P, Q, R are structural equivalent to D', P', Q', R', as well as the queues. The parallel composition of processes is associative in TyCO (Rule ($\textsc{Assoc}$ on page 29)).

Sending objects

$$\begin{array}{ccc}
\eta \cup \{x\langle B/\rangle\}, M & \asymp & \eta \cup \{y\langle C/\rangle\}, M \quad (x, y \notin \{fn(\eta) \cup fn(B)\}) \\
\downarrow & & \downarrow \\
\textbf{def } D \textbf{ in } (P|Q) & \equiv_T & \textbf{def } D' \textbf{ in } (P'|Q')
\end{array}$$

The structural equivalence of D, P, Q and $D', P'Q'$ can be shown in the same way as in the case of structural equivalence. The difference is that the names of the process type calculus objects are not equal.

This means, that all the object does, is create new objects, send messages or change its behavior (but this behavior still receives no answers). Therefore, the identity of the object is not important.

Now we prove $\eta, M \asymp \theta, N \Longleftarrow P \equiv_T Q$. As before, we consider twelve cases:

α-conversion: Is also supported by the process type calculus.

Commutative and associative parallel composition: The programs programs are translated to objects in our calculus. The rules concerning object commutativity and associativity are similar in our calculus.

Transformations of the $\mathbf{0}$ process: The TyCO programs we create do not include empty processes.

Commutativity of scope restriction:

$$\begin{tikzcd}
(\nu xy)Q \arrow[d] & \equiv_T & (\nu yx)Q \arrow[d] \\
\eta \cup \{x\langle B_1/\mu\rangle, y\langle B_2/\nu\rangle\}, M & \asymp & \eta' \cup \{y\langle B_1'/\mu\rangle, x\langle B_2'/\nu\rangle\}, M'
\end{tikzcd}$$

The x and y are the object identifiers and are equal on both sides. The behavior B_1 corresponds to B_1' on the left hand side, as well as B_2 corresponds to B_2'. The message queues are equal (see Rule (Send on page 57) and Rule (Object on page 57)).

Extension of scope restriction:

$$\begin{tikzcd}
(\nu x)Q|R \arrow[d] & \equiv_T & (\nu x)(Q|R) \arrow[d] & (x \notin fn(Q)) \\
\eta \cup \{x\langle B/\tilde{\mu}\rangle\}, M & \asymp & \eta' \cup \{x\langle B'/\tilde{\mu}\rangle\}, M' &
\end{tikzcd}$$

Extension of scope restriction in a definition:

$$\begin{tikzcd}
\mathbf{def}\ D\ \mathbf{in}(\nu x)Q \arrow[d] & \equiv_T & (\nu x)\ \mathbf{def}\ D\ \mathbf{in}\ Q \arrow[d] & (x \notin fn(Q)) \\
\eta\cup\{y\langle B/\tilde{\mu}\rangle\}, M & \asymp & \eta'\cup\{y'\langle B'/\tilde{\mu}\rangle\}, M &
\end{tikzcd}$$

On the left hand side, the configuration contains an object y behaving like $(\nu x)Q$, a placeholder for x and n creator methods l_i representing D. On the right hand side, the configuration contains c, l_j, which behave like Q and D and the object d representing a placeholder for x.

Extension of a definition: TyCO Rule (Def_3 on page 29) obviously holds in the process type calculus because the creator methods representing D are globally visible.

Inflating definitions:

$$\begin{tikzcd}
\mathbf{def}\ D\ \mathbf{in}\ \mathbf{def}\ D'\ \mathbf{in}\ Q \arrow[d] & \equiv_T & \mathbf{def}\ D\ \mathbf{and}\ D'\ \mathbf{in}\ Q \arrow[d] \\
\eta, M\cup\{l_i(\underline{\tilde{x}}_i)B_i, l_j(\underline{\tilde{x}}_j)B_j\} & \asymp & \eta', M'\cup\{l_k(\underline{\tilde{x}}_k)B_k, l_l(\underline{\tilde{x}}_l)B_l\}
\end{tikzcd}$$

$$(1\leq i, k\leq m \leq j, l\leq n)$$

The behaviors of the objects and creator methods on both sides are corresponding to each other.

Definition application:

$$\begin{tikzcd}
\mathbf{def}\ X = A\ \mathbf{and}\ D\ \mathbf{in}\ X[\tilde{v}]|Q \arrow[d] & \equiv_T & \mathbf{def}\ X = A\ \mathbf{and}\ \mathbf{def}\ D'\ \mathbf{in}\ A[\tilde{v}]|Q \arrow[d] \\
\eta \cup \{a\langle @l(\tilde{v})/\rangle\}, N \cup \{l(\underline{\tilde{x}})B\} & \asymp & \eta \cup \{b\langle @l'\tilde{v})/\rangle\}, N \cup \{l'(\underline{\tilde{x}}')B'\}
\end{tikzcd}$$

On the left hand side, the behavior B corresponds to A (The parameters are obtained by processing A with Rule ($\triangle$ APL on page 51)) and an object a is created which behaves like B. On the right hand side, the same configuration is created, with the difference, that the Rule ($\triangle$ APL) is called a second time in order to translate $A[\tilde{v}]$.

Instantiation:

$$\begin{array}{ccc} ((\tilde{x})P)[\tilde{v}] & \equiv_T & P[\tilde{v}/\tilde{x}] \\ \downarrow & & \downarrow \\ \eta \cup \{a\langle @l(\tilde{v})/\rangle\}, M \cup \{l(\underline{\tilde{x}})B\} & \asymp & \eta' \cup \{b\langle B'/\rangle\}, M' \end{array}$$

The abstract translation in rule (instance) calls rule ($\triangle$ agent). These rules return a label with a fresh name l and a behavior. The creator method l can be replaced using rule ($\text{call}_{\rightarrow}$).

Commutativity of methods: A collection of methods is translated to a process type method selection which may also be permuted.

Commutativity of definitions: The definitions are translated into creator methods which are put in a set, therefore, the ordering does not matter.

□

Theorem 12 *The algorithm given in Figure 4.5 on page 57 terminates.*

PROOF. Configurations in the process type calculus are structured in the following way: $x_1\langle B_1/\tilde{\mu}_1\rangle, .., x_n\langle B_n/\tilde{\mu}_n\rangle, l_1(\underline{\tilde{x}}_1)B_1, .., l_n(\underline{\tilde{x}}_n)B_n$. First, the $l_i(\underline{\tilde{x}}_i)B_i$ are handled according to Rule (CREATOR on page 57). Every creator method is mapped to a **def** statement. The behavior of the creator method is a finite sequence of select, send or new operations ended by a become operation. Each of these operations is handled with one rule.

Objects $l_1(\underline{\tilde{x}}_1)B_1, .., l_n(\underline{\tilde{x}}_n)B_n$ are translated using Rule (OBJECT). The behavior of an object is a finite sequence of select, send or new operations ended by a become operation. Each of these operations is handled with one rule. The message queue is a finite sequence of messages which are inserted to the queue by Rule (OBJECT). The algorithm is obviously deterministic. Therefore, the algorithm is linear to the number of objects, their messages in queues, and the number of creator methods. □

Theorem 13 *The core process type calculus has the same expressive power as the TyCO calculus.*

PROOF. Follows immediately from Definition 44 on page 52 and the theorems 9 and 11. □

4.6 Discussion

Among the large variety of proposed calculi for concurrent systems, two different models of communication can be identified:

- The channel based model, like the π-calculus [MPW92, Mil92] and its relatives, for example [Bou97] and TyCO [Vas99]. Communication is made via channels, and objects send messages to and get messages from a channel. There may be more than one reader from a channel. It can not be guaranteed that a message is read by a specific object.

- The actor based model, like the actor model [HBS73, Agh86, AMST92] and its relatives [GZ99a, NN99]. Communication is made directly from object to object. Each object has a unique identifier. Therefore, a message is always read by the object it is sent to.

In the process type calculus, each object has its own identifier and messages that are received but not processed are kept in a queue. We have chosen to compare TyCO with our calculus because it incorporates the features of the π-calculus and also supports some higher level constructs like abstraction. The type system of TyCO is representative for conventional type systems. And as it can be fully abstract encoded to the π-calculus, all we state for TyCO also holds for the π-calculus. The fully abstract encoding we gave from the core process type calculus to TyCO and vica versa is proving that all we can express in TyCO also can be expressed in the process type calculus. In the following chapter we will examine types for the process type calculus and TyCO.

Chapter 5

The Typed Process Type Calculus

5.1 Process Types

In this chapter we will introduce types for the process type calculus. We also allow types to be passed as parameters. The type parameters can be used to provide parametric polymorphism. This feature can be used to provide genericity like templates in C++ [Str97]. Another use of this mechanism is to support the polymorphic typing assignments used in TyCO (See Section 5.5.2 on page 74).

This is the syntax of types (denoted by $\pi, \rho, \sigma, \tau, \ldots$):

$$\begin{aligned}
\tau &::= \mathbf{\{}\tilde{\alpha}\mathbf{\}}[\tilde{a}] \quad | \quad (\underline{t})\mathbf{\{}\tilde{\alpha}\mathbf{\}}[\tilde{a}] \quad | \quad \tau_1|\tau_2 \quad | \quad t \\
\alpha &::= x(\tilde{\underline{s}})(\tilde{\tau})[\tilde{a}_1][\tilde{a}_2] \\
a &::= x \quad | \quad x^{\infty}
\end{aligned}$$

A type of the form $\mathbf{\{}\tilde{\alpha}\mathbf{\}}[\tilde{a}]$ consists of a set of message descriptors (each denoted by $\alpha, \beta, \gamma, \ldots$) and a multi-set of tokens $(a, b, c, \ldots)$. We call $\mathbf{\{}\tilde{\alpha}\mathbf{\}}$ the static part and $[\tilde{a}]$ the dynamic part of the type. An object's type changes dynamically by removing tokens from and adding tokens to the type's dynamic part when messages are accepted.

A token is either a simple token x or an infinite token x^{∞} standing for a very large (or infinite) number of simple tokens x. We use x^n as an abbreviation of n occurrences of a simple token x in a multi-set.

Message descriptors are of the form $x(\tilde{\underline{s}})(\tilde{\tau})[\tilde{a}][\tilde{c}]$, where x is the message selector, $\tilde{s}$ are type parameters and $\tilde{\tau}$ types of object parameters. The type parameters can occur in the object parameters' types. Message selector, type parameters and parameter types together are considered to be the signature of a message. The other parts of a message descriptor are the in-set $[\tilde{a}]$ and the out-set $[\tilde{c}]$. A message of the specified signature is acceptable (by an object of a type containing the message descriptor in its static part) if each token in the in-set occurs also in the dynamic part of the object's type. When accepting the message, the type is updated according to this message descriptor by removing all tokens contained in the in-set from the type's dynamic part and adding all tokens contained in the out-set to the dynamic part.

A type of the form $\sigma|\tau$ denotes the parallel composition of two types; $\sigma|\tau$ can be split into σ and τ.

A type of the form $(\underline{t})\{\tilde{\alpha}\}[\tilde{a}]$ is a recursive version of the type. The type parameter t can (recursively) occur in $\tilde{\alpha}$. A type parameter can occur in each position where a type can occur.

The static part of a window's type can be expressed by:

$$\mathit{Window} \stackrel{\text{def}}{=} \{\text{iconify}()()[\text{displ}][\text{icon}], \text{uniconify}()()[\text{icon}][\text{displ}]\}$$

When a window receives a message "iconify", a token "displ" is removed from the type's dynamic part, and "icon" is added. When receiving "uniconify", "icon" is removed and "displ" added. An object of type Window[displ] first accepts "iconify", then "uniconify", then "iconify" again, and so on.

We distinguish between the type of an object and the type of a reference to an object. The type of an object describes all messages accepted by the object. The type of a reference describes the messages that can be sent to the object via this reference. It may contain only a part of the object's type information. The combination of the types of all references to an object must keep the contract that the type of the object specifies. This can be ensured with the mechanism of "type splitting": When a new reference (alias) is created, the type information is split in an old part (associated with an already existing reference) and a new part (associated with the new reference). For example, a reference's type Window[displ] can be split into Window[displ] and Window[], but not into Window[displ] and Window[displ] (because there is just a single token "displ").

It is possible for a client to follow the changes of an object visible through a reference by keeping track of the messages that were sent via this reference. For example, when a message is passed via a reference of type Window[displ], the reference will be updated to Window[icon]. Type updating can be performed statically by means of abstract interpretation.

Windows usually understand further messages like "move" to change the location of the displayed window or icon, and "redisplay" to redisplay the possibly scrambled contents of the displayed window. A window with these extensions can have the type

$$\mathit{Window}[\text{displ}] \,|\, \{\text{move}()(\mathit{location})[][], \text{redisplay}()()[\text{displ}][\text{displ}]\}$$

where *location* is the type of an object specifying the location on the screen. The message "move" is accepted in arbitrary interleaving with the other messages. But "redisplay" is accepted only if the window is displayed. A client with a reference of type mark Window[displ] can send only "iconify" and "uniconify" (in alternation) to the window. A client with a reference of a type mark equal to the right of "|" can send only "move" because the state known by the client does not contain [displ]; "redisplay" can be sent only after combining the two type marks.

The static part of a simple heterogenous buffer is:

$$\begin{aligned} \mathit{Buffer} &\stackrel{\text{def}}{=} \{\text{put}(\underline{t})(t)[\text{empty}][\text{full}], \text{get}()(\mathit{Back}[\text{one}])[\text{full}][\text{empty}]\} \\ \mathit{Back} &\stackrel{\text{def}}{=} \{\text{back}(\underline{t})(t)[\text{one}][]\} \end{aligned}$$

A message "put" is accepted if a buffer slot is empty, and "get" if a slot is full; the empty slot becomes full, and the full slot empty, respectively. "Put" has a type parameter and

$$P ::= a\langle B/\tilde{\mu}\rangle \qquad \mu ::= l(\tilde{\tau})(\tilde{v}) \qquad M ::= l(\underline{\tilde{s}})(\underline{\tilde{x}})B$$
$$B ::= \{\tilde{M}\} \quad | \quad a{\triangleleft}\mu.B \quad | \quad (\underline{v})@\mu.B \quad | \quad @\mu$$

Figure 5.1: Syntax of objects with type parameters

an object parameter; an object reference with a type mark equal to the type provided as first argument is expected as second argument. "Get" expects as arguments a reference to an object accepting a single message "back"; it is assumed that the contents of a buffer slot as well as its type are provided as arguments of "back". *Buffer*[full] is the type of an empty buffer with a single slot, *Buffer*[full^{50}] that of an empty buffer with fifty slots, and *Buffer*[empty$^{\infty}$] that of an empty buffer with unlimited number of slots.

The behavior partially specified by the last type is specified also by *BufferI*[] with:

$$\mathit{BufferI} \stackrel{\text{def}}{=} \{\mathrm{put}(\underline{t})(t)[\,][\mathrm{full}], \mathrm{get}()(\mathit{Back}[\mathrm{one}])[\mathrm{full}][\,]\}$$

An object of this type always accepts "put", but "get" only if there is a full slot. Objects of the types *Buffer*[empty$^{\infty}$] and *Buffer*[full$^{\infty}$, empty$^{\infty}$] accept "put" and "get" in arbitrary ordering. The syntax of objects presented in Figure 5.1 supports type parameters. It extends Figure 4.1 on page 46 with type parameters which can now be communicated via messages. A message $(\mu, \nu, \xi, \dots)$ of the form $x(\tilde{\tau})(\tilde{v})$ associates a message selector x with a (possibly empty) list of types $\tilde{\tau}$ and a (possibly empty) list of object identifiers or object parameters $\tilde{v}$ used as actual parameters. The types $\tilde{\tau}$ can be used in partially specifying the behavior of the objects with identifiers $\tilde{v}$. Dynamically passing types as parameters provides a flexible kind of genericity as shown below by examples.

A method $(M, N, \dots)$ of the form $x(\underline{\tilde{s}})(\underline{\tilde{v}})B$ specifies the behavior B preceded by a guard defining a method name (or message selector) x and the formal parameters $\tilde{s}$ and $\tilde{v}$.

We have to change the rules for a-execution given in Definition 42 on page 47 so that they support type parameters:

Definition 45 (a-execution relation) *The a-execution relation $\eta_{\{\tilde{M}\}} \xrightarrow{a} \theta_{\{\tilde{M}\}}$ on system configurations is the least closure of these rules:*

$$\frac{\tilde{P} \equiv \tilde{P}' \quad \eta_{\{\tilde{M}\}} \cup \{\tilde{P}'\} \xrightarrow{a} \eta_{\{\tilde{M}\}} \cup \{\tilde{Q}'\} \quad \tilde{Q}' \equiv \tilde{Q}}{\eta_{\{\tilde{M}\}} \cup \{\tilde{P}\} \xrightarrow{a} \eta_{\{\tilde{M}\}} \cup \{\tilde{Q}\}} \qquad (\textsc{Congr}_{\rightarrow})$$

$$\eta_{\{\tilde{M}\}} \cup @l(\tilde{s})(\tilde{v}) \xrightarrow{a} \eta_{\{\tilde{M}\}} \cup B \quad (l(\underline{\tilde{s}})(\underline{\tilde{v}})B \in \{\tilde{M}\}) \qquad (\textsc{Call}_{\rightarrow})$$

$$\eta_{\{\tilde{M}\}} \cup a\langle \{l(\underline{\tilde{s}})(\underline{\tilde{v}})B, \tilde{N}\}/l(\tilde{\tau})(\tilde{u}), \tilde{\mu}\rangle \xrightarrow{a} \eta_{\{\tilde{M}\}} \cup a\langle B[\tilde{\tau}/\tilde{s}][\tilde{u}/\tilde{v}]/\tilde{\mu}\rangle \qquad (\textsc{Select}_{\rightarrow})$$

$$\eta_{\{\tilde{M}\}} \cup a\langle a{\triangleleft}\xi.B/\tilde{\mu}\rangle \xrightarrow{a} \eta_{\{\tilde{M}\}} \cup a\langle B/\tilde{\mu}, \xi\rangle \qquad (\textsc{Self send}_{\rightarrow})$$

$$\eta_{\{\tilde{M}\}} \cup \{x\langle y{\triangleleft}\xi.B/\tilde{\mu}\rangle, y\langle C/\tilde{\nu}\rangle\} \xrightarrow{a} \eta_{\{\tilde{M}\}} \cup \{x\langle B/\tilde{\mu}\rangle, y\langle C/\tilde{\nu}, \xi\rangle\} \qquad (\textsc{Send}_{\rightarrow})$$

$$\eta_{\{\tilde{M}\}} \cup \{x\langle(\underline{v})@\xi.B/\tilde{\mu}\rangle\} \xrightarrow{a} \eta_{\{\tilde{M}\}} \cup \{x\langle B[y/v]/\tilde{\mu}\rangle, y\langle @\xi[y/v]/\rangle\} \qquad (\textsc{New}_{\rightarrow})$$

The execution relation $\eta_{\{\tilde{M}\}} \xrightarrow{*} \theta_{\{\tilde{M}\}}$ is the least reflexive and transitive closure of all a-execution relations $\eta_{\{\tilde{M}\}} \xrightarrow{a} \theta_{\{\tilde{M}\}}$ with $a \in \mathbb{X}$.

The next example was inspired by Milner [MPW92]: A car with a mobile phone moves between two base stations. When the car moves from one base to the other, a center initiates a hand-over procedure. We require that always only one base communicates with the car. The following creator methods are used to implement the example:

$$\begin{aligned}&\{\mathrm{Car}()(\underline{bs})\{\mathrm{talk}()()\ldots@\mathrm{Car}(bs),\ \mathrm{switch}(\underline{bs'})@\mathrm{Car}()(bs')\},\\&\mathrm{Base}()(\underline{car})\{\mathrm{idle}()(\underline{bs'})\,car\triangleleft\mathrm{switch}()(bs').@\mathrm{Idlebase}()(car),\\&\qquad\qquad\qquad\mathrm{talk}()()\,car\triangleleft\mathrm{talk}()().@\mathrm{Base}()(car)\},\\&\mathrm{Idlebase}()(\underline{car})\{\mathrm{alert}()()@\mathrm{Base}(car)\},\\&\mathrm{Center}()(\underline{bs},\underline{bs'})\ldots bs\triangleleft\mathrm{idle}()(bs').bs'\triangleleft\mathrm{alert}()()\ldots@\mathrm{Center}()(bs,bs')\}\end{aligned}$$

The following objects are also part of the configuration

$$\{\mathrm{car}\langle@\mathrm{Car}(bs)/\rangle,\mathrm{bs}\langle@\mathrm{Base}(car)/\rangle,\mathrm{bs}'\langle@\mathrm{Idlebase}(car)/\rangle,\mathrm{ctr}\langle@\mathrm{Center}(bs,bs')/\rangle)\}$$

A base is in one of two states, active or idle. The active base behaves as Base()($\underline{car}$), the other as Idlebase()($\underline{car}$). When the center tells the active base to become idle, the base informs the car about the switch, alerts the other base and then becomes idle. An idle base waits for an alert message and then becomes active. The center decides when to switch bases.

We define the following types:

$$\begin{aligned}CarT &\stackrel{\text{def}}{=} \{\mathrm{talk}()()[][],\ \mathrm{switch}()(BaseT[])[][]\}\\BaseT &\stackrel{\text{def}}{=} \{\mathrm{talk}()()[\mathrm{active}][\mathrm{active}],\ \mathrm{idle}()(BaseT[])[\mathrm{active}][\mathrm{idle}],\\&\qquad\mathrm{alert}()()[\mathrm{idle}][\mathrm{active}]\}\end{aligned}$$

The type $CenterT$ for the Center is omitted, because we have not defined its behavior. The tokens in $BaseT$ ensure that always at most one base station is active.

5.2 Equivalence for Process Types

Process types partially specify object behavior as sets of all acceptable message sequences. Two process types are regarded as equivalent if they specify the same set of acceptable message sequences.

Before we can define type equivalence, we need further notions:

For each name x and set $\{\tilde{\alpha}\}$ of message descriptors, $del_x(\tilde{\alpha})$ denotes a set of message descriptors equal to $\{\tilde{\alpha}\}$, except that all simple tokens x and infinite tokens x^∞ are removed from the in-sets and out-sets of $\tilde{\alpha}$.

Two types σ and τ are structurally equivalent, denoted by $\sigma \equiv \tau$, if they can be made equal by renaming bound type parameters (α-conversion) and repeatedly applying these rules to static and dynamic type parts:

$$[\tilde{a},b,\tilde{c}] \equiv [\tilde{a},\tilde{c},b] \qquad [x^\infty,x,\tilde{a}] \equiv [x^\infty,\tilde{a}] \qquad [x^\infty,x^\infty,\tilde{a}] \equiv [x^\infty,\tilde{a}]$$

$$\{\tilde{\alpha},\beta,\tilde{\gamma}\} \equiv \{\tilde{\alpha},\tilde{\gamma},\beta\} \qquad \{\tilde{\alpha},\beta,\beta\} \equiv \{\tilde{\alpha},\beta\}$$

$$\Pi \vdash \sigma \cong \tau \quad (\sigma \equiv \tau) \tag{Struct$_\cong$}$$
$$\Pi \vdash (\rho|\sigma)|\tau \cong \rho|(\sigma|\tau) \tag{Assoc$_\cong$}$$
$$\Pi \vdash \sigma|\tau \cong \tau|\sigma \tag{Commut$_\cong$}$$
$$\Pi \vdash \tau|\mathbf{\{\}}[] \cong \tau \tag{Empty$_\cong$}$$
$$\Pi \vdash \mathbf{\{}\tilde{\alpha}\mathbf{\}}[\tilde{a}]|\mathbf{\{}\tilde{\gamma}\mathbf{\}}[\tilde{c}] \cong \mathbf{\{}\tilde{\alpha},\tilde{\gamma}\mathbf{\}}[\tilde{a},\tilde{c}] \quad (\{\tilde{a}\} \subseteq \mathit{rel}(\tilde{\alpha});\ \{\tilde{c}\} \subseteq \mathit{rel}(\tilde{\gamma})) \tag{Par$_\cong$}$$
$$\Pi \vdash (\underline{t})\mathbf{\{}\tilde{\alpha}\mathbf{\}}[\tilde{a},b] \cong (\underline{t})\mathbf{\{}\tilde{\alpha}\mathbf{\}}[\tilde{a}] \quad (b \notin \mathit{rel}(\tilde{\alpha})) \tag{Nonrel$_\cong$}$$
$$\Pi \vdash (\underline{t})\mathbf{\{}\tilde{\alpha}\mathbf{\}}[\tilde{a},x^\infty] \cong (\underline{t})\mathbf{\{}\tilde{\gamma}\mathbf{\}}[\tilde{a}] \quad (\{\tilde{\gamma}\} = \mathit{del}_x(\tilde{\alpha})) \tag{Avail$_\cong$}$$
$$\Pi \vdash (\underline{t})\mathbf{\{}\tilde{\alpha},\tilde{\gamma}\mathbf{\}}[\tilde{a}] \cong (\underline{t})\mathbf{\{}\tilde{\alpha}\mathbf{\}}[\tilde{a}] \quad (\forall_{\gamma\in\{\tilde{\gamma}\}}\,\exists_{\alpha\in\{\tilde{\alpha}\}}\,\Pi,\mathit{rel}(\tilde{\alpha}) \vdash \alpha \gg \gamma) \tag{Red$_\cong$}$$
$$\Pi \vdash (\underline{t})\mathbf{\{}\tilde{\alpha}\mathbf{\}}[\tilde{a}] \cong \mathbf{\{}\tilde{\alpha}[(\underline{t})\mathbf{\{}\tilde{\alpha}\mathbf{\}}[\tilde{a}]/t]\mathbf{\}}[\tilde{a}] \tag{Rec$_\cong$}$$

Figure 5.2: Type Equivalence

Definition 46 (Type equivalence) *The type equivalence relation $\Pi \vdash \sigma \cong \tau$ on types depending on a subtyping environment Π is the least congruence closed under the rules in Figure 5.2. We write $\sigma \cong \tau$ instead of $\Pi \vdash \sigma \cong \tau$ if Π is empty.*

The redundancy relation $\Pi, X \vdash \alpha \gg \gamma$ on message descriptors, depending on a subtyping environment Π and a set X of tokens, is defined by this rule:

$$\Pi, X \vdash x(\underline{\tilde{s}})(\tilde{\sigma})[\tilde{a}][\tilde{c},\tilde{c}'] \gg x(\underline{\tilde{s}})(\tilde{\tau})[\tilde{a},\tilde{a}',\tilde{a}''][\tilde{a}'',\tilde{c},\tilde{c}'']$$

where $\Pi \vdash \tilde{\tau} \leq \tilde{\sigma}$ (as defined below) and $\{\tilde{c}''\} \cap X = \emptyset$.

Parallel composition $|$ is an associative and commutative operation with the empty type $\mathbf{\{\}}[]$ as neutral element. As specified by (par$_\cong$), two types of the forms $\mathbf{\{}\tilde{\alpha}\mathbf{\}}[\tilde{a}]$ and $\mathbf{\{}\tilde{\gamma}\mathbf{\}}[\tilde{c}]$ are composed to $\mathbf{\{}\tilde{\alpha},\tilde{\gamma}\mathbf{\}}[\tilde{a},\tilde{c}]$ by creating the unions of the message descriptor sets and the multi-sets of tokens. All tokens in the dynamic parts of the composed types must be relevant. Nonrelevant tokens can be eliminated by using (nonrel$_\cong$). Rule (avail$_\cong$) allows us to remove tokens that are always available in the dynamic type part (and, therefore, do not constrain the acceptability of messages) simultaneously from the dynamic part and the message descriptors' in-sets and out-sets.

Applications of (red$_\cong$) can remove redundant message descriptors from static type parts. A message descriptor is redundant if there is another message descriptor that can be used instead of the redundant one. The nonredundant message descriptor removes at most those tokens from the dynamic type part that the redundant message descriptor removes when an instance accepts a message corresponding to both message descriptors, and the nonredundant message descriptor adds at least those relevant tokens added by the redundant message descriptor. Rule (red$_\cong$) removes several message descriptors simultaneously because tokens may become nonrelevant by removing several message descriptors, but not by removing just a single one.

Rule (rec$_\cong$) deals with folding and unfolding recursive types. An important special case of this rule is $\Pi \vdash (\underline{t})\mathbf{\{}\tilde{\alpha}\mathbf{\}}[\tilde{a}] \cong \mathbf{\{}\tilde{\alpha}\mathbf{\}}[\tilde{a}]$ if t does not occur free in $\tilde{\alpha}$.

An example for type equivalence is *BufferI*$[] \cong$ *Buffer*$[\mathrm{empty}^\infty]$, which can be shown using (avail$_\cong$).

5.3 Mechanisms for Typechecking

A process type consists of two parts: A set of signatures and creators and a state. Each signature specifies the conditions for accepting corresponding messages depending on the state. It also specifies the way the state is changed after a message is accepted. A creator specifies the signature of a message creating a new instance of the type and the initial state of the new object. The basic operations on process types are:

- *Type updating*: The state of a type is updated when sending or accepting messages. The tokens in the in-set are removed from the multi-set representing the state, the tokens in the out-set are added to the state.
- *Type splitting*: A type is split into two types (being supertypes of the first type) such that all arbitrary interleavings of message sequences specified by one of the supertypes with those specified by the other supertypes are specified by the first type also.
- *Type combination*: This is the reverse operation of type splitting. Two types are combined to a single one, being a subtype of both of the two types.

Process types can be used in two different ways:

- *Object type*: Each object is associated with a type partially specifying the object's behavior. The object must accept all message sequences specified by the type. The type is updated when the object accepts a message. Type splitting and type combination are not applied to object types.
- *Type mark*: Each reference to an object is associated with a type mark which specifies all sequences of messages that can be sent to the object via this reference. The type mark is updated when a message is sent via the reference. When a new reference is created, type splitting is used and the composition of the new references' type mark must be equal to the old references' type mark. A type mark can be regarded as a claim to send messages. It can be used up (by sending messages) or given via type splitting to another object (in part or as whole) as side effect of sending messages. The programmer has to specify how the type mark is split by specifying which object references are passed as arguments. In this way the programmer specifies how the clients are coordinated.

5.4 Type Checking Rules for Process Types

Figure 5.3 on the following page lists the type checking rules. An expression of the form $\Gamma \vdash \langle B{:}\tau \rangle$ means that the behavior B is compatible with the partial behavior specified by the type τ (this is an object with current behavior B is of type τ), depending on a typing environment Γ. This environment contains typings of the forms $v{:}\sigma$ and $x(\underline{\tilde{s}})(\tilde{\pi}){:}\sigma$. A typing of the first form says that v is an object parameter; the corresponding actual parameter is of type σ. All typings of this form in a typing environment shall have

$$\frac{\Gamma \cup \{u{:}\sigma|\pi\} \vdash \langle B{:}\tau|\pi\rangle}{\Gamma \cup \{u{:}\sigma|\rho\} \vdash \langle B{:}\tau|\rho\rangle} \quad (\tau|\pi \text{ and } \tau|\rho \text{ deterministic}) \qquad (\textsc{Self})$$

$$\frac{\forall_{i=1..n}\, \Gamma \cup \{\tilde{v}_i{:}\tilde{\tau}_i\} \vdash \langle B_i{:}\boldsymbol{\{}\tilde{\alpha}\boldsymbol{\}}[\tilde{c}_i]\rangle}{\Gamma \vdash \langle \boldsymbol{\{} x_1(\underline{\tilde{s}}_1)(\underline{\tilde{v}}_1)B_1, \ldots, x_n(\underline{\tilde{s}}_n)(\underline{\tilde{v}}_n)B_n \boldsymbol{\}}{:}\boldsymbol{\{}\tilde{\alpha}\boldsymbol{\}}[\tilde{a}]\rangle} \quad (1) \qquad (\textsc{Select})$$

$$\frac{\Gamma \cup \{v{:}\boldsymbol{\{}x(\underline{\tilde{s}})(\tilde{\pi})[\tilde{a}][\tilde{c}]\boldsymbol{\}}[\tilde{c}]|\sigma\} \vdash \langle \tilde{w}{:}\tilde{\pi}[\tilde{\rho}/\tilde{s}], B{:}\tau\rangle}{\Gamma \cup \{v{:}\boldsymbol{\{}x(\underline{\tilde{s}})(\tilde{\pi})[\tilde{a}][\tilde{c}]\boldsymbol{\}}[\tilde{a}]|\sigma\} \vdash \langle v{\triangleleft}x(\tilde{\rho})(\tilde{w}).B{:}\tau\rangle} \qquad (\textsc{Send})$$

$$\frac{\Gamma \cup \{x(\underline{\tilde{s}})(\tilde{\pi}){:}\sigma, v{:}\sigma[\tilde{\rho}/\tilde{s}]\} \vdash \langle \tilde{w}{:}\tilde{\pi}[\tilde{\rho}/\tilde{s}], B{:}\tau\rangle}{\Gamma \cup \{x(\underline{\tilde{s}})(\tilde{\pi}){:}\sigma\} \vdash \langle (\underline{v})@x(\tilde{\rho})(\tilde{w}).B{:}\tau\rangle} \qquad (\textsc{New})$$

$$\frac{\Gamma \cup \{x(\underline{\tilde{s}})(\tilde{\pi}){:}\sigma\} \vdash \langle \tilde{w}{:}\tilde{\pi}[\tilde{\rho}/\tilde{s}], \boldsymbol{\{\}}{:}\tau\rangle}{\Gamma \cup \{x(\underline{\tilde{s}})(\tilde{\pi}){:}\sigma\} \vdash \langle @x(\tilde{\rho})(\tilde{w}){:}\sigma[\tilde{\rho}/\tilde{s}]|\tau\rangle} \qquad (\textsc{Call})$$

$$\frac{\Gamma \cup \{v{:}\pi\} \vdash \langle \tilde{w}{:}\tilde{\sigma}, B{:}\tau\rangle}{\Gamma \cup \{v{:}\pi|\rho\} \vdash \langle v{:}\rho, \widetilde{w{:}\tilde{\sigma}}, B{:}\tau\rangle} \qquad (\textsc{Param})$$

(1) $\boldsymbol{\{}\tilde{\alpha}\boldsymbol{\}}[\tilde{a}]$ deterministic; $\{x_1, \ldots, x_n\} = \{x \mid x(\tilde{s})(\tilde{\tau})[\tilde{c}] \in act(\boldsymbol{\{}\tilde{\alpha}\boldsymbol{\}}[\tilde{a}])\}$;
$x_i(\tilde{s}_i)(\tilde{\tau}_i)[\tilde{c}_i] \in act(\boldsymbol{\{}\tilde{\alpha}\boldsymbol{\}}[\tilde{a}])$;

Figure 5.3: Type checking rules

pairwise different object parameters. A typing of the second form says that the program executed by the object system contains a creator method of signature $x(\underline{\tilde{s}})(\tilde{\pi})$; the behavior of this method is partially specified by the type σ, where the type parameters $\tilde{s}$ can occur in σ. All typings of this form in a typing environment shall have pairwise different method names.

By convention, the object parameter u used in Figure 5.3 is the self-reference in behavior B. Other object references (in $\mathbb{V}\backslash\mathbb{X}$) are denoted by $v, w, \ldots$. Object identifiers shall not occur in B.

An expression of the form $\Gamma \vdash \langle v_1{:}\sigma_1, \ldots, v_n{:}\sigma_n, B{:}\tau\rangle$ or, equivalently, $\Gamma \vdash \langle \tilde{v}{:}\tilde{\sigma}, B{:}\tau\rangle$ specifies that additional checks (possibly causing modifications of Γ) must be performed before checking $B{:}t$. Checking $v_i{:}\sigma_i$ ensures that the object parameter v_i is of the required type σ_i. As specified by (param), the type of each actual parameter is split (along parallel composition) into two types; the type corresponding to the formal parameter's type is removed, and the other type remains in the typing environment.

Let us have a look at the type checking rules. The type associated with a self-reference and the type partially specifying a behavior B may be changed simultaneously in the same way, as specified by (self). If a type can be split off (at a parallel composition) from the self-reference's type, messages specified by this type can no longer be sent through the self-reference. Since objects behaving according to B need no longer to be able to accept these messages, the same type can be split off from the objects' type. On the other side of $\vdash$, if type information is added to the type partially specifying B, an object of this type can deal with all additional messages specified by this type. It is safe to add the same type information to the self-reference's type.

The side condition of (select) ensures that there is a method for each message that may be received. When checking the methods' behaviors, the methods' formal parameters can be used; Γ is extended correspondingly.

Rule (send) ensures that the receiver of a message is already known and will accept the message (according to its type). The receiver's type is changed as specified by the message descriptor in the type's static part. Furthermore, the actual object parameters are checked to be of appropriate types.

For each invocation of a creator method, (call) and (new) check if there actually is a creator method of appropriate signature and behavior. The actual object parameters are checked to be of appropriate types.

Definition 47 (Consistency) *A set $\{\tilde{M}\}$ of creator methods (considered to be a program) is type-consistent with respect to a typing environment Γ if*

- *each expression in Γ is of the form $x(\tilde{\underline{s}})(\tilde{\sigma}){:}\tau$, and for each such expression there is a creator method $x(\tilde{\underline{s}})(\tilde{v})B \in \{\tilde{M}\}$ with $|\tilde{\sigma}| = |\tilde{v}|$,*
- *for each $x(\tilde{\underline{s}})(\tilde{v})B \in \{\tilde{M}\}$ there is an expression $x(\tilde{\underline{s}})(\tilde{\sigma}){:}\tau \in \Gamma$ such that $\Gamma \cup \{\tilde{v}^x{:}\tilde{\sigma}\} \vdash \langle B{:}\tau\rangle$ as specified by the rules in Figure 5.3 on the preceding page.*

The second condition of this definition ensures that each creator method in a program behaves as partially specified by the expressions in the typing environment. The corresponding type checks (usually checked by a compiler) can be performed separately for each creator method. Thus, we have separate compilation. The first condition (usually checked by a linker) ensures that there are creator methods implemented for all expressions occurring in the typing environment.

We will resume the example given in Section 5.1 on page 63 and describe the typing environment. We define types for the creator methods:

$$\begin{array}{l}
\mathrm{Car}()(\mathit{BaseT}[]){:}\boldsymbol{\{}\mathrm{talk}()()[][],\ \mathrm{switch}(\mathit{BaseT}[])[][]\boldsymbol{\}}[] \\
\mathrm{Base}()(\mathit{CarT}[]){:}\boldsymbol{\{}\mathrm{idle}()(\mathit{BaseT}[\mathrm{idle}])[\mathrm{active}][\mathrm{idle}], \mathrm{talk}()()[\mathrm{active}][\mathrm{active}]\boldsymbol{\}}[\mathrm{active}] \\
\mathrm{Idlebase}()(\mathit{CarT}[]){:}\boldsymbol{\{}\mathrm{alert}()()[\mathrm{idle}][\mathrm{active}]\boldsymbol{\}}[\mathrm{idle}] \\
\mathrm{Center}()(\mathit{BaseT}[\mathrm{active}], \mathit{BaseT}[\mathrm{idle}]){:}\boldsymbol{\{}\ldots\boldsymbol{\}}[]
\end{array}$$

The typings associated to the objects given in the example are:

$$\{\mathrm{car{:}CarT}[], \mathrm{bs{:}BaseT}[\mathrm{active}], \mathrm{bs'{:}BaseT}[\mathrm{idle}], \mathrm{Ctr{:}CenterT}[\ldots]\}$$

5.5 A Comparison with the TyCO Type System

TyCO programs are typed by a type inference system (see Section 3.4 on page 30). Now we want to translate TyCO process typings of the form $B, \Gamma \vdash A : \alpha$ to object typings in the process type calculus $\Gamma \vdash B : \pi$. A syntactic difference is that in TyCO a typing assignment consists of a base B of type assignments to abstraction variables and typings to names which are stored in Γ_T. The typing in the process type calculus contains typings for creator methods and objects in Γ.

$$C \equiv \{l_{i1}(\underline{\tilde{t}}_1)(\underline{\tilde{x}}_1)B_1, .., l_{in}(\underline{\tilde{t}}_n)(\underline{\tilde{x}}_n)B_n\}$$

$$\mathrm{rcv}()()\{\mathrm{r}l_{i1..in}()(\underline{y})@\mathrm{send}()(y), l_1(\underline{\tilde{t}}_1)(\underline{\tilde{x}}_1)\mathit{self}\triangleleft l_1(\tilde{x}_1).@\mathrm{rcv}()(), \ldots\}$$

$$\mathrm{send}()(\underline{y})\{l_{i1}(\underline{\tilde{t}}_1)(\underline{\tilde{x}}_1).y\triangleleft l_{i1}(\underline{\tilde{t}}_1)(\tilde{x}_1).@\mathrm{rcv}()(), \ldots, \mathrm{r}l_{i1..in}()(\underline{y})\mathit{self}\triangleleft \mathrm{r}l_{i1..in}(y)@\mathrm{send}()()\}$$

$$\tau' = \mathrm{r}l_{i1..in}(\underline{\tilde{t}})(\tilde{\pi})[\,][\,], \mathrm{r}l_{i1..in}(\tilde{\pi})[\,][\,], l_1(\tilde{\pi}_1)[\,][\,], .., l_n(\tilde{\pi}_n)[\,][\,], l_1(\tilde{\pi}_1)[\,][\,], .., l_n(\tilde{\pi}_n)[\,][\,]$$

$$\frac{\eta, M \vdash P \rightarrow \theta, N \quad \eta, M \vdash Q \rightarrow \theta', N'}{\eta \vdash P|Q \rightarrow \theta \cup \theta', N \cup N'} \qquad (\mathrm{TC{\scriptsize ON}})$$

$$\eta, M \vdash a!l_i{:}\tilde{v} \rightarrow \eta \cup \{b\langle a\triangleleft l_i(\tilde{s})(\tilde{v}).\{\}/\rangle{:}\{\}[\,]\}, M \quad (1) \qquad (\mathrm{TS{\scriptsize ND}})$$

$$\frac{\forall_i(\eta \cup \{a\langle C'/\mu\rangle{:}\tau'[\,]\}, M \vdash \triangle A_i \rightarrow l_i(\underline{\tilde{t}}_i)(\underline{\tilde{x}}_i)B_i{:}\pi_i, N)_{1\leq i\leq n}}{\eta \cup \{a\langle C'/\mu\rangle{:}\tau'[\,]\}, M \vdash a?\{l_{i1}{:}A_1\&\cdot\cdot\} \rightarrow \eta \cup \{b\langle C/\rangle{:}\{\pi_1, ..\}[\,], a\langle C'/\mu, r_{i1...in}()(b)\rangle{:}\tau'[\,]\}, M}$$

$$(\Gamma_T(a) \approx_T \{l_1{:}\tilde{\alpha}_1, ..\}) \qquad (\mathrm{TS{\scriptsize EL}})$$

$$\frac{\eta \cup \{a\langle @\mathrm{rcv}()()/\rangle{:}\tau'[\,]\}, M \vdash P[a/x] \rightarrow \theta, N}{\eta, M \vdash (\nu x)P \rightarrow \theta, N} \quad (a\,\mathrm{fresh}), (\Gamma_T \backslash x) \qquad (\mathrm{TR{\scriptsize ES}})$$

$$\eta, M \vdash (\nu x)\mathbf{0} \rightarrow \eta, M \qquad (\mathrm{TR{\scriptsize ES}}\mathbf{0})$$

$$\frac{(\eta, M_{i-1} \vdash \triangle A_i \rightarrow l_i(\underline{\tilde{s}}_i)(\tilde{x}_i)B_i{:}\pi_i, M_i \cup \{l_i(\underline{\tilde{s}}_i)(\tilde{x}_i)B_i{:}\pi_i\})_{1\leq i\leq n} \quad \eta, M_n \vdash (Q[l_i/X_i])_{1\leq i\leq n} \rightarrow \theta, N}{\eta, M_0 \vdash \mathbf{def}\ X_1 = A_1\ \mathbf{and} \ldots \mathbf{and}\ X_n = A_n\ \mathbf{in}\ Q \rightarrow \theta, N}$$

$$(\mathrm{TD{\scriptsize EF}})$$

$$\eta, M \vdash \mathbf{def}\ X_1 = A_1\ \mathbf{and} \ldots \mathbf{and}\ X_n = A_n\ \mathbf{in}\ \mathbf{0} \rightarrow \eta, M \qquad (\mathrm{TD{\scriptsize EF}}\mathbf{0})$$

$$\eta, M \cup \{l(\underline{\tilde{s}})(\underline{\tilde{y}})B{:}\tilde{\pi}\} \vdash (\tilde{y})l[\tilde{x}] \rightarrow \eta \cup \{a\langle @l(\tilde{s}\theta_T)(\tilde{x})/\rangle{:}\tilde{\pi}\}, M \cup \{l(\underline{\tilde{s}})(\underline{\tilde{y}})B{:}\tilde{\pi}\} \quad (2) \qquad (\mathrm{TV{\scriptsize AR}})$$

$$\frac{\eta, M \vdash \triangle(\tilde{y})P \rightarrow l(\underline{\tilde{s}})(\underline{\tilde{y}})B{:}\tilde{\pi}, N \cup \{l(\underline{\tilde{s}})(\underline{\tilde{y}})B{:}\tilde{\pi}\}}{\eta, M \vdash A[\tilde{x}] \rightarrow \eta \cup \{a\langle @l(\tilde{s}\theta_T)(\tilde{x})/\rangle{:}\tilde{\pi}\}, N \cup \{l(\underline{\tilde{s}})(\underline{\tilde{y}})B{:}\tilde{\pi}\}} \quad (2) \qquad (\mathrm{TA{\scriptsize PP}})$$

$$\eta, M \vdash \ \mathbf{0} \rightarrow \eta, M \qquad (\mathrm{TN{\scriptsize UL}})$$

Figure 5.4: Typed process construction rules

5.5.1 Monomorphic Type System

The rules to translate well-typed TyCO programs to well-process typed configurations are given in Figure 5.4 and Figure 5.5 on the following page. $\Gamma_T(a)$ denotes the TyCO type of a, $\Gamma_T \setminus x$ is obtained by removing the type assignment for x from Γ_T (see Section 3.4.1 on page 30). θ_T is a unification of TyCO terms.

The rules are a typed version of the rules in Figure 4.3 and Figure 4.4 on page 51. The typed process construction rules generate typed objects and creator methods which are typed as well. When an abstraction is defined, the corresponding typed behavior is provided by Rule ($\triangle$ TABS), and a typed creator method with the behavior is inserted into the environment.

The typed agent construction rules do not generate typed behavior, but they generate typed creator methods or methods. To construct typings Rule ($\triangle$ TABS) is used.

Rules (TRES**0**), (TDEF**0**), (TNUL), ($\triangle$ TRES**0**), ($\triangle$ TDEF**0**) and ($\triangle$ TNUL) do not construct any processes or creator methods. In order to use the features of genericity that are automatically supported by type inference systems (it is possible to infer a typing

$$\frac{\eta, M \vdash \triangle P \to B, N \quad \eta, M \vdash \triangle Q \to B', N'}{\eta, M\vdash\triangle P|Q\to(\underline{x})@l_1()().(\underline{y})@l_2()().\mathbf{\{\}}, N\cup N'\cup\{l_1()()B{:}l_1()(), l_2()()B'{:}l_2()()\}} \quad (1) \qquad (\triangle\ \text{TCon})$$

$$\eta, M \vdash a!\triangle l{:}\tilde{v} \to a\triangleleft l(\tilde{s})(\tilde{v}), M \quad (2) \qquad (\triangle\ \text{TSnd})$$

$$\frac{\forall_i(\eta, M \vdash \triangle A_j \to B_j, N_j)_{1\le j\le n}}{\eta, M\vdash\triangle a?\{l_{i1}{:}A_1\&\cdot\cdot\& l_{in}{:}A_n\}\to(\underline{b})@l()().a\triangleleft r_{i1\ldots in}(b).\mathbf{\{\}}, \bigcup_{1\le j\le n} N_j\cup\{l()C'{:}\tau'[]\}} \qquad (\triangle\ \text{TSel})$$

$$\frac{\eta, M\cup\{l()()@rcv()():\tau'[]\}\vdash\triangle P\to B,N}{\eta,M\vdash\triangle(\nu x)P\to(\underline{x})@l()().B,N} \qquad (\triangle\ \text{TRes})$$

$$\eta, M\vdash\triangle(\nu x)\ \mathbf{0}\to\mathbf{\{\}}, M \qquad (\triangle\ \text{TRes}\mathbf{0})$$

$$\frac{(\eta, M_{i-1}\vdash\triangle A_i\to l_i(\underline{\tilde{s}_i})(\tilde{x}_i)B_i{:}\tilde{\pi}_i, M_i\cup\{l_i(\underline{\tilde{s}_i})(\tilde{x}_i)B_i{:}\tilde{\pi}_i\})\quad \eta, M_n\vdash(\triangle Q[l_i/X_i])_{1\le i\le n}\to B', N}{\eta, M_0\vdash \mathbf{def}\ X_1 = A_1\ \mathbf{and}\ldots\mathbf{and}\ X_n = A_n\ \mathbf{in}\ Q\to B', N} \qquad (\triangle\ \text{TDef})$$

$$\eta, M\vdash\triangle\mathbf{def}\ X_1=A_1\ \mathbf{and}\ldots\mathbf{and}\ X_n=A_n\ \mathbf{in}\ \mathbf{0}\to\mathbf{\{\}}, M \qquad (\triangle\ \text{TDef}\mathbf{0})$$

$$\eta, M\cup\{l(\underline{\tilde{s}})(\underline{\tilde{y}})B{:}\tilde{\pi}\}\vdash l[\tilde{x}]\to(\underline{x})@l(\underline{\tilde{s}}\theta_T)(\tilde{x}).\mathbf{\{\}}, M\cup\{l(\underline{\tilde{s}}')(\tilde{y})B{:}\tilde{\pi}\}\quad(2) \qquad (\triangle\ \text{TVar})$$

$$\frac{\eta,M\vdash\triangle(\tilde{y})P\to l(\underline{\tilde{s}})(\underline{\tilde{x}})B{:}\tilde{\pi},N}{\eta,M\vdash\triangle A[\tilde{x}]\to(\underline{x})@l(\tilde{s}\theta_T)(\tilde{x}).\mathbf{\{\}}{:}\{\}[],N\{l(\underline{\tilde{s}})(\underline{\tilde{x}})B{:}\tilde{\pi}\}}\quad(3) \qquad (\triangle\ \text{TApp})$$

$$\frac{\eta,M\vdash\triangle P\to B{:}\tilde{\pi},N}{\eta,M\vdash\triangle(\tilde{x})P\to l(\underline{\tilde{s}})(\underline{\tilde{x}})B{:}l(\underline{\tilde{s}})(\tilde{\alpha})[][], N\cup\{l(\underline{\tilde{s}})(\underline{\tilde{x}})B{:}l(\underline{\tilde{s}})(\tilde{\alpha})[][]\}}\quad(l\text{ fresh},(1)) \qquad (\triangle\ \text{TAbs})$$

$$\eta,M\vdash\triangle\mathbf{0}\to\mathbf{\{\}},M \qquad (\triangle\ \text{TNul})$$

$$(1)\quad (l_1,l_2\text{ fresh}) \qquad (2)\quad \Gamma_T(\tilde{x})\approx_T\tilde{\alpha},\forall t_i\in\mathbb{V}, t_i\in fn(\tilde{\alpha})\underline{\tilde{s}}{=}t_1,t_2\ldots)$$

$$(3)\quad \tilde{y}\approx_T\tilde{x}\Rightarrow\tilde{y}\theta_T\equiv_T\tilde{x}\theta_T$$

Figure 5.5: Typed Agent construction rules

$a{:}t$ where t is a type variable). We have to pass types as parameters in the process type calculus. The syntax of a TyCO send operation (the receive operation is corresponding) is $a!l{:}\tilde{v}$, where the $\tilde{v}$ are of type $\tilde{\alpha}_i$. The $\tilde{\alpha}_i$ may also contain type variables t_i, we have to map them in $(\tilde{s})$. We use unification to bind the t_i to $\tilde{s}$ in rules (TVar), (TApp), ($\triangle$ TVar), ($\triangle$ TApp) and ($\triangle$ TAbs).

The typing in Rule (TVar) is created by Rule (TDef) which uses Rule ($\triangle$ TAbs) to get typed creator methods.

An abstraction Rule (TApp) is typed using Rule ($\triangle$ TAbs).

When a message $a\triangleleft l(\tilde{s})(\tilde{v})$ is sent, the s have to be constructed according to the TyCO type information. The typing of objects that only do a send operation is $\{\}[]$ (Rule (TSnd) and Rule ($\triangle$ TSnd)).

A TyCO restriction is translated to a placeholder object in the process type calculus. The typing of this placeholder object is given in τ' (Rule (TRes) and Rule ($\triangle$ TRes)).

Selections are dealt with in (Rule (TSel) and Rule ($\triangle$ TSel)). The typings for the agents of a selection is received by calling Rule ($\triangle$ TAbs) for every possible label of the selection. Then the typings for these labels are added to the typing of the selection.

TyCO programs that shall be executed concurrently are handled independently, their typings are merged unchanged (Rule (TCon) and Rule ($\triangle$ TCon)).

Definitions are using Rule ($\triangle$ TAbs) to get a typed creator method and then insert this typed creator method to the configuration (Rules TDef and $\triangle$ TDef).

Theorem 14 *All well-typed TyCO programs can be translated to well-typed programs in the typed process type calculus.*

PROOF. The proof shall be done by case analysis:

Typings for methods are created by Rule ($\triangle$ TAbs). The type parameters ($\underline{\tilde{s}}$) are the free type variables that are used in the TyCO typing. The ($\tilde{\alpha}$) are the types of the TyCO typing (see side condition (1)). The inset and the outset of the method remain empty. Therefore, all the conditions of the type checking rule Select are fulfilled. When a message is sent, the parameters ($\tilde{v}$) remain the same, but the type variables are sent in ($\tilde{s}$) (see Rules (TSnd) and ($\triangle$ TSnd)). The conditions in Rule (Select) and Rule (Send) are satisfied. The object created by Rule (TSnd) has only the empty type, as specified in Rule (Send).

The next typing assumption is made when abstractions are applied, which is translated to the call of a creator method. When an abstraction is applied, the actual ($\tilde{x}$) and the formal parameters ($\tilde{y}$) are unified (side condition (2) of rules TVar, TApp, $\triangle$ TVar, $\triangle$ TApp). The resulting unificator θ_T is applied to ($\tilde{s}$). Again, the type checking conditions for a call are satisfied.

The object created by restriction has type τ', which can easily be checked using rules (Select), (Send) and (Call). The types of the methods in the selection statement are created by the abstraction rule, and these types are added to the type of the object. The rule for concurrency does not affect the typing of the configuration. $\square$

Theorem 15 *The algorithm given in Figure 5.4 on page 71 and Figure 5.5 on the preceding page terminates.*

PROOF. The algorithm is a typed version of the algorithm in Figure 4.3 on page 49 and Figure 4.4 on page 51 whose termination is stated in Theorem 10 on page 56. $\square$

The translation of typed process type expressions into TyCO is not possible. For example a one-place buffer with the following creator methods

$$\mathrm{Bempty}()()\{\mathrm{put}(\underline{t})(\underline{el})@\mathrm{Bufferfull}(\underline{t})(\underline{el})\}$$
$$\mathrm{Bfull}(\underline{t})(\underline{el})\{\mathrm{get}()(\underline{rcv})\,rcv\triangleleft\mathrm{back}()(el).@\mathrm{Bufferempty}()()\}$$

which may store one element of arbitrary type, has the typings given on Page 5.1 on page 64 and can not be typed correctly in TyCO, because it would contain one object which would want to receive a “put” once and then a “get” which is not correct in TyCO.

5.5.2 Polymorphic Type System

The polymorphic type system for TyCO allows to type processes with abstractions that are applied using different types. An example with the corresponding typing is given in 3.4.2 on page 32. This property is included in the version of process types given in the previous section as every creator method is called with type parameters. When a creator method is called more than once, it may be called with different type parameters. Therefore, process type programs have the same expressiveness as the polymorphic type system for TyCO.

5.5.3 Ravara and Vasconcelos' Behavioral Types for TyCO

The notion of the behavioral types for TyCO defined by Ravara and Vasconcelos [RV97] is comparable to process types. The types contain states and the sending of messages results in a state change. But, the difference between process types and the behavioral types used by Ravara and Vasconcelos is the notion of error (see Definition 29 on page 34): A process is erroneous if there is no possibility to send a correct message to the process. For example, the process $S \stackrel{\text{def}}{=} \text{Empty}(x)|x!\text{read}:[u]$ is not regarded as an error because the message $x!\text{write}:[v]$ could be sent to x sometimes and then the message $x!\text{read}:[u]$ would be correct. Process types do not allow an equivalent configuration (containing an object $\{x\langle\text{@Empty/read(u)}\rangle\}$ and creator methods $\{\text{Empty}()\boldsymbol{\{}\text{write}(\underline{u})\text{@Full}(b,u)\boldsymbol{\}}, \text{Full}(u)\text{read}(r)r\triangleleft(u).\text{@Empty}\})$. Ravara and Vasconcelos' types allow reordering of messages which is not allowed in the process type calculus. Therefore, the concept of errors is more "optimistic" than the error concepts in TyCO and process types.

5.6 Subtyping for Process Types

According to Liskov, a type is a partial specification of object behavior [LW93, LW94]. A subtype specifies the same behavior as its supertypes, but in more detail. Subtyping is characterized by the principle of substitutability: An instance of a subtype can be used where an instance of a supertype is expected, without affecting type safety.

A process type is a subtype of another process type if each message sequence specified by the supertype is also specified by the subtype. Furthermore, type equivalence and subtyping shall satisfy this extensibility condition: If σ is equivalent with (or a subtype of) τ, then $\sigma|\rho$ shall be equivalent with (or a subtype of) $\tau|\rho$ for each type ρ.

A subtyping environment Π is a (possibly empty) set of subtyping assumptions of the form $s \leq t$ (the type parameter s is a subtype of the type parameter t).

Definition 48 (Subtyping) *The subtyping relation $\Pi \vdash \sigma \leq \tau$ on types, depending on a subtyping environment Π, is the least reflexive and transitive relation closed under the rules in Figure 5.6 on the next page. We write $\sigma \leq \tau$ instead of $\Pi \vdash \sigma \leq \tau$ if Π is empty.*

$$\Pi \cup \{s \leq t\} \vdash s \leq t \qquad (\text{ASSUMPTION}_{\leq})$$

$$\frac{\Pi \vdash \sigma \cong \tau|\rho}{\Pi \vdash \sigma \leq \tau} \qquad (\text{SUB}_{\leq})$$

$$\frac{\Pi \vdash \pi \leq \rho \quad \Pi \vdash \sigma \leq \tau}{\Pi \vdash \pi|\sigma \leq \rho|\tau} \qquad (\text{PAR}_{\leq})$$

$$\frac{\Pi \cup \{s \leq t\} \vdash \boldsymbol{\{}\tilde{\alpha}\boldsymbol{\}}[\tilde{a}] \leq \boldsymbol{\{}\tilde{\gamma}\boldsymbol{\}}[\tilde{c}]}{\Pi \vdash (\underline{s})\boldsymbol{\{}\tilde{\alpha}\boldsymbol{\}}[\tilde{a}] \leq (\underline{t})\boldsymbol{\{}\tilde{\gamma}\boldsymbol{\}}[\tilde{c}]} \; (s \notin \mathit{fn}(\tilde{\gamma}, \Pi);\; t \notin \mathit{fn}(\tilde{\alpha}, \Pi)) \qquad (\text{REC}_{\leq})$$

Figure 5.6: Subtyping

A type σ is a subtype of a type τ if there is a type ρ such that σ and the parallel composition of τ and ρ are equivalent. The type ρ specifies the difference between σ and τ. As a special case, $\Pi \vdash \sigma \cong \tau$ implies $\Pi \vdash \sigma \leq \tau$. $\Pi \vdash \sigma \leq \boldsymbol{\{\}}[]$ always holds.

The type checking rules for the process type calculus given in Section 5.4 on page 68 can be extended to allow contravariant arguments in methods.

We can now allow substitution of types with their supertypes. The rules (SUB SERVER) and (SUB CLIENT) are added to the rules in Figure 5.3 on page 69.

$$\frac{\Gamma \vdash \langle B{:}\sigma\rangle}{\Gamma \vdash \langle C{:}\tau\rangle} \; (B \cong_{\emptyset} C;\; \sigma \leq \tau;\; \sigma \text{ and } \tau \text{ deterministic}) \qquad (\text{SUB SERVER})$$

$$\frac{\Gamma \cup \{v^x{:}\rho\} \vdash \langle B{:}\tau\rangle}{\Gamma \cup \{v^x{:}\sigma\} \vdash \langle B{:}\tau\rangle} \; (\sigma \leq \rho) \qquad (\text{SUB CLIENT})$$

There are two such rules because an object is always checked from two views: From the client view and from the server view. Rule (sub server) allows the typing of a behavior to be replaced by supertype, Rule (sub client) allows the same with a client.

For example, $\mathit{Buffer}[\text{empty}^{m+m'}, \text{full}^{n+n'}]$ is a subtype of $\mathit{Buffer}[\text{empty}^{m}, \text{full}^{n}]$, where $\mathit{Buffer}[\text{empty}^{m'}, \text{full}^{n'}]$ is the difference between the two types. Type $\mathit{BufferI}[\text{full}^{n+n'}]$ is a subtype of $\mathit{Buffer}[\text{empty}^{m}, \text{full}^{n}]$, where $\mathit{BufferI}[\text{full}^{n'}]$ is the difference between subtype and supertype.

Theorem 16 *Type equivalence and subtyping are decidable for finite (but possibly recursive) types.*

We sketch the proof only briefly. The congruence rules for type equivalence can be reorganized as a commutative and associative rewrite system. A type is in normal form if no rewrite rule is applicable on it. If there exists a normal form for a type, all applications of rewrite rules lead to the same normal form. Two types are equivalent if and only if they have the same normal form.

Decidability of subtyping in conventional type systems has been addressed by Amadio and Cardelli in [AC91]. The proof of decidability of subtyping in our calculus goes along the lines of their proof. The major difference from their notion of subtyping is rule (sub$_{\leq}$): We have to find a type ρ with $\Pi \vdash \sigma \cong \tau|\rho$. For σ of the form $\boldsymbol{\{}\tilde{\alpha}\boldsymbol{\}}[\tilde{a}]$, we can always select $\rho = \boldsymbol{\{}\tilde{\alpha}\boldsymbol{\}}[\tilde{c}]$, where the $\tilde{c}$ are computed as aside effect of applying (par$_{\leq}$). The other cases are rather simple.

Subtyping based on a typing environment Π is decidable if type equivalence based on Π is decidable; and type equivalence based on Π is decidable if subtyping of parameter types is decidable. Thus, subtyping and type equivalence are decidable for types with a limited number of parameter levels. As in Amadio and Cardelli's proof, it is not necessary to infinitely expand recursive types. $\square$

As in the above examples, the static part of a type giving the difference between subtype and supertype can always be selected so that it is equal to the static part of the subtype. Hence, it is easy to decide if there is a type ρ as required by (sub$_\leq$).

A type is deterministic if it is equivalent to a type of the form $\boldsymbol{\{}\tilde{\alpha}\boldsymbol{\}}[\tilde{a}]$ such that all message descriptors $\tilde{\alpha}$ have pairwise different message selectors.

The type of each object shall be deterministic to ensure that, whenever a message is accepted, there is only one way to change the type's dynamic part. Users of objects know supertypes of these objects' types. Although the supertypes need not be deterministic, each change of the dynamic part of one of these supertypes when a user sends a message causes a corresponding change of an object's deterministic type when the message is accepted. Thus, the sender of a message knows in advance how the receiver's type will change when the message is accepted.

A message signature $x(\underline{\tilde{s}})(\tilde{\tau})$ is active with a follow-state $[\tilde{c}]$ in a type $\boldsymbol{\{}\tilde{\alpha}\boldsymbol{\}}[\tilde{a}]$ if $x(\underline{\tilde{s}})(\tilde{\tau})[\tilde{c}] \in \mathit{act}(\boldsymbol{\{}\tilde{\alpha}\boldsymbol{\}}[\tilde{a}])$, where act is defined by

$$\mathit{act}(\boldsymbol{\{}\tilde{\alpha}\boldsymbol{\}}[\tilde{a}]) = \{x(\underline{\tilde{s}})(\tilde{\tau})[\tilde{a}', \tilde{c}'] \mid x(\underline{\tilde{s}})(\tilde{\tau})[\tilde{c}][\tilde{c}'] \in \{\tilde{\alpha}\};\ [\tilde{a}] \equiv [\tilde{a}', \tilde{c}]\}.$$

If a message signature is active with follow-state $[\tilde{c}]$ in a type $\boldsymbol{\{}\tilde{\alpha}\boldsymbol{\}}[\tilde{a}]$, an object of this type accepts a (single) message of this signature. When the message is accepted, the type is changed to $\boldsymbol{\{}\tilde{\alpha}\boldsymbol{\}}[\tilde{c}]$. The signatures of the messages acceptable next are active in the changed type.

Next we introduce sequences of message prototypes (denoted by $p, q, \dots$) that specify message sequences rather directly. Types can to a large part be represented by sets of such sequences. A message prototype (denoted by $\varphi, \dots$) is an expression of the form $x(T, \dots, T)(\tilde{p})$, where each message prototype sequence p is of the form $\langle\tilde{\varphi}\rangle$. The (infinite) set T contains all possible message prototype sequences. A message prototype $x(T, \dots, T)(\tilde{p})$ is regarded as representing messages $x(\tilde{\tau})(\tilde{y})$, where each T stands for an arbitrary optional type in $\tilde{\tau}$, and each y_i in $\tilde{y}$ is the identifiers of an object accepting a sequence of messages as represented by p_i through the corresponding message parameter. The function $\mathit{seq}(\tau)$ returns the message prototype set specified by the type τ:

$$\begin{aligned}
\mathit{seq}(\tau_1|\dots|\tau_i|\dots|\tau_n) &= T \quad (n \geq 1;\ \tau_i \in \mathbb{T}) \\
\mathit{seq}(\tau_1|\dots|\tau_i|\dots|\tau_n) &= \mathit{seq}(\tau_1|\dots|\boldsymbol{\{}\tilde{\alpha}[(\underline{t})\boldsymbol{\{}\tilde{\alpha}\boldsymbol{\}}[\tilde{a}]/t]\boldsymbol{\}}[\tilde{a}]|\dots|\tau_n) \\
&\qquad\qquad (n \geq 1;\ \tau_i = (\underline{t})\boldsymbol{\{}\tilde{\alpha}\boldsymbol{\}}[\tilde{a}]) \\
\mathit{seq}(\tau_1|\dots|\tau_n) &= \mathit{seq}(\boldsymbol{\{}\tilde{\alpha}_1, \dots, \tilde{\alpha}_n\boldsymbol{\}}[\tilde{a}_1, \dots, \tilde{a}_n]) \\
&\qquad (\tau_i \equiv \boldsymbol{\{}\tilde{\alpha}_i\boldsymbol{\}}[\tilde{a}_i, \tilde{c}_i];\ \{\tilde{a}_i\} \subseteq \mathit{rel}(\tilde{\alpha}_i);\ \{\tilde{c}_i\} \not\subseteq \mathit{rel}(\tilde{\alpha}_i)) \\
\mathit{seq}(\boldsymbol{\{}\tilde{\alpha}\boldsymbol{\}}[\tilde{a}]) &= \{\langle\tilde{\varphi}\rangle \mid \langle\tilde{\varphi}\rangle[\tilde{c}] \in \bigcup_{i\geq 0} A_i(\boldsymbol{\{}\tilde{\alpha}\boldsymbol{\}}[\tilde{a}]);\}
\end{aligned}$$

where the A_i are defined by:

$$\begin{aligned} A_0(\{\tilde\alpha\}[\tilde a]) &= \{\langle\rangle[\tilde a]\} \\ A_{i+1}(\{\tilde\alpha\}[\tilde a]) &= \{\langle\tilde\varphi, x(\tilde T)(\tilde p)\rangle[\tilde c'] \mid \langle\tilde\varphi\rangle[\tilde c] \in A_i;\ x(\underset{\sim}{s})(\tilde\tau)[\tilde c'] \in \mathit{act}(\{\tilde\alpha\}[\tilde c]); \\ &\qquad |\tilde T| = |\tilde s|;\ p_j \in \mathit{seq}(\tau_j);\ 1 \le j \le |\tilde\tau|\} \end{aligned}$$

Theorem 17 *For all types σ and τ, if $\sigma \cong \tau$, then $\mathit{seq}(\sigma) = \mathit{seq}(\tau)$.*

We sketch the proof: Most rules defining $\cong$ on types obviously do not influence whether or not a message signature is active. Only ($\mathrm{red}_{\cong}$) is a bit more difficult. Let σ be of the form $\{\tilde\alpha\}[\tilde a]$ and τ of the form $\{\tilde\gamma\}[\tilde c]$; and let ($\mathrm{red}_{\cong}$) be applied to σ and τ such that $[\tilde a] \equiv [\tilde c, \tilde c']$ for some $\tilde c'$ with $\tilde c' \notin \mathit{rel}(\tilde\gamma)$, and $\{\tilde\alpha\} \equiv \{\tilde\gamma, \tilde\beta\}$ for some $\tilde\beta$; the $\tilde\beta$ are removed by applying ($\mathrm{red}_{\cong}$). Hence, $\mathit{seq}(\tau) \subseteq \mathit{seq}(\sigma)$. The side condition of ($\mathrm{red}_{\cong}$) implies that message descriptors are deleted only if there remain more general message descriptors. The remaining message descriptors are active whenever the removed ones were active. This implies $\mathit{seq}(\sigma) \subseteq \mathit{seq}(\tau)$ and $\mathit{seq}(\sigma) = \mathit{seq}(\tau)$. □

Theorem 18 *For all types σ and τ, if $\sigma \le \tau$, then $\mathit{seq}(\tau) \subseteq \mathit{seq}(\sigma)$.*

First, we show a property of sequences and parallel compositions. If there are types $\{\tilde\alpha\}[\tilde a]$, π and ρ with $\{\tilde\alpha\}[\tilde a] \cong \pi|\rho$, then $p \otimes q \subseteq \mathit{seq}(\{\tilde\alpha\}[\tilde a])$ for each $p \in \mathit{seq}(\pi)$ and $q \in \mathit{seq}(\rho)$, where $p \otimes q$ denotes the set of all arbitrary interleavings of the sequence p and q. This is easy to see from the definition of *seq*.

Let $\sigma \le \tau$. If there is a type ρ with $\sigma \cong \tau|\rho$ according to ($\mathrm{sub}_{\le}$), then $\mathit{seq}(\rho)$ is prefix-closed (because ρ is optional) and contains the empty sequence $\langle\rangle$. If σ is equivalent to a type $\{\tilde\alpha\}[\tilde a]$, then $\mathit{seq}(\tau) \subseteq \mathit{seq}(\sigma)$ follows from $p \otimes \langle\rangle \subseteq \mathit{seq}(\{\tilde\alpha\}[\tilde a])$ for all $p \in \mathit{seq}(\tau)$. Otherwise σ contains a type parameter at the top level (not as a parameter type) and $\mathit{seq}(\tau) \subseteq \mathit{seq}(\sigma)$ simply follows from $\mathit{seq}(\sigma) = T$. The proof for recursive types goes along the lines of a similar proof in [AC91]. □

Obviously, the reverses of the above theorems do not hold: $\mathit{seq}(\sigma) = \mathit{seq}(\tau)$ does not imply $\sigma \cong \tau$, and $\mathit{seq}(\tau) \subseteq \mathit{seq}(\sigma)$ does not imply $\sigma \le \tau$. For example, two different type parameters are not equivalent or related by subtyping (if the subtyping environment is empty), although both corresponding sequence sets are equal to T. As another example, the types $\{x()()[x][]\}[]$ and $\{y()()[y][]\}[]$ (with $x \ne y$) are not equivalent, although $\mathit{seq}(\{x()()[x][]\}[]) = \{\langle\rangle\} = \mathit{seq}(\{y()()[y][]\}[])$. But, our notions of type equivalence and subtyping are complete for types not containing free type parameters when considering the extensibility condition:

Theorem 19 *Let σ and τ be types with $\mathit{fn}(\sigma) = \emptyset = \mathit{fn}(\tau)$. Then, $\sigma \cong \tau$ if and only if $\mathit{seq}(\sigma|\rho) = \mathit{seq}(\tau|\rho)$ for all types ρ.*

One direction of the proof, $\sigma \cong \tau$ implies $\mathit{seq}(\sigma|\rho) = \mathit{seq}(\tau|\rho)$ for all types ρ, immediately follows from Theorem 17. The other direction can be shown by a case analysis on all reasons for $\sigma \not\cong \tau$, although $\mathit{seq}(\sigma) = \mathit{seq}(\tau)$: A type ρ not satisfying $\mathit{seq}(\sigma|\rho) = \mathit{seq}(\tau|\rho)$ is constructed for each reason. Since free type parameters do not occur in the types and the rules ($\mathrm{par}_{\cong}$), ($\mathrm{nonrel}_{\cong}$) and ($\mathrm{rec}_{\cong}$) are always applicable, we can assume, without restricting generality, that σ and τ are of the forms $\{\tilde\alpha\}[\tilde a]$ and $\{\tilde\gamma\}[\tilde c]$, respectively, with $\tilde a \in \mathit{rel}(\tilde\alpha)$ and $\tilde c \in \mathit{rel}(\tilde\gamma)$.

- If σ and τ differ in the constant names used as tokens, there is a ρ containing a message descriptor such that the acceptability of corresponding messages depends on the availability of tokens available in σ or τ, but not in both simultaneously.

- If $\{\tilde{\alpha}\}$ and $\{\tilde{\gamma}\}$ contain different message descriptors, but some of them are never active because required tokens never become available, appropriate ρ make these tokens available.

- If tokens are relevant in σ but not in τ (or vice versa) and these tokens do not represent a limited resource because they are always available, a message descriptor in ρ can remove these tokens so that they become a limited resource. (There is no such ρ if a token x^∞ occurs in the type's dynamic part; but then (avail$_\cong$) is applicable.)

- If σ (or τ) contains an active redundant message descriptor not occurring in τ (or σ), there is a message descriptor $x(\underline{\tilde{s}})(\tilde{\sigma})[\tilde{a}'][\tilde{c}']$ in $(\{\tilde{\alpha}\} \cup \{\tilde{\gamma}\}) \setminus (\{\tilde{\alpha}\} \cap \{\tilde{\gamma}\})$ and a message descriptor $x(\underline{\tilde{s}})(\tilde{\tau})[\tilde{a}''][\tilde{c}'']$ in $\{\tilde{\alpha}\} \cap \{\tilde{\gamma}\}$ with $\tilde{\sigma} \leq \tilde{\tau}$, both message descriptors being used in the construction of an A_i, but the first (redundant) message descriptor does not satisfy the side condition of (red$_\cong$) because $[\tilde{a}']$ and $[\tilde{c}']$ are not related appropriately with $[\tilde{a}'']$ and $[\tilde{c}'']$. There are types ρ that introduce tokens such that $x(\underline{\tilde{s}})(\tilde{\tau})[\tilde{a}''][\tilde{c}'']$ is not always active when $x(\underline{\tilde{s}})(\tilde{\sigma})[\tilde{a}'][\tilde{c}']$ is active, or message descriptors in ρ become active by adding $\tilde{c}'$ to (or removing $\tilde{a}'$ from) the types' dynamic parts, but not by adding $\tilde{c}''$ (or removing $\tilde{a}''$). □

For example, there is a type $\rho = \{z()()[z][x]\}[z]$ so that $seq(\{x()()[x][]\}[]|\rho) = \{\langle\rangle, \langle z()()\rangle, \langle z()(), x()()\rangle\}$ and $seq(\{y()()[y][]\}[]|\rho) = \{\langle\rangle, \langle z()()\rangle\}$.

Theorem 20 *Let σ and τ be types with $fn(\sigma) = \emptyset = fn(\tau)$. Then, $\sigma \leq \tau$ if and only if $seq(\tau|\rho) \subseteq seq(\sigma|\rho)$ for all types ρ.*

The first direction of the proof, $\sigma \leq \tau$ implies $seq(\tau|\rho) \subseteq seq(\sigma|\rho)$ for all types ρ, immediately follows from Theorem 18 on the page before. The other direction can be shown by a case analysis on all reasons for $\sigma \not\leq \tau$, although $seq(\sigma) \subset seq(\tau)$ (real subset relationship because $seq(\tau) = seq(\sigma)$ was already dealt with by Theorem 19). A type ρ not satisfying $seq(\tau|\rho) \subseteq seq(\sigma|\rho)$ is constructed for each reason. Without restricting generality, let $\sigma = \{\tilde{\alpha}\}[\tilde{a}]$ and $\tau = \{\tilde{\gamma}\}[\tilde{c}]$ with $\tilde{a} \in rel(\tilde{\alpha})$ and $\tilde{c} \in rel(\tilde{\gamma})$. The relationship $\sigma \leq \tau$ does not hold if and only if there are no tokens $\tilde{c}'$ with $\{\tilde{\alpha}\}[\tilde{a}] \cong \{\tilde{\alpha}, \tilde{\gamma}\}[\tilde{c}, \tilde{c}']$.

- A message descriptor γ in $\{\tilde{\gamma}\}$ with signature $x(\underline{\tilde{r}})(\tilde{\sigma})$ cannot be removed by applying (red$_\cong$) because there is no message descriptor in $\{\tilde{\alpha}\}$ with a signature $x(\underline{\tilde{s}})(\tilde{\tau})$, where $\tilde{\sigma} \leq \tilde{\tau}[\tilde{r}/\tilde{s}]$; the message descriptor γ is never active. There is a type $\rho = \{\tilde{\gamma}'\}[\tilde{c}']$ introducing tokens such that γ becomes active and $seq(\{\tilde{\gamma}, \tilde{\gamma}'\}[\tilde{c}, \tilde{c}']) \not\subseteq seq(\{\tilde{\alpha}, \tilde{\gamma}'\}[\tilde{a}, \tilde{c}'])$.

- A message descriptor $x(\underline{\tilde{r}})(\tilde{\sigma})[\tilde{a}'][\tilde{c}']$ in $\{\tilde{\gamma}\}$ cannot be removed by applying (red$_\cong$) because there is no message descriptor $y(\underline{\tilde{s}})(\tilde{\tau})[\tilde{a}''][\tilde{c}'']$ in $\{\tilde{\alpha}\}$ such that $[\tilde{a}']$ is related with $[\tilde{a}'']$ and $[\tilde{c}']$ with $[\tilde{c}'']$ as required by (red$_{\cong}$). There is a type $\rho = \{\tilde{\gamma}'\}[\tilde{c}''']$ introducing tokens such that message descriptors in ρ become active by adding $\tilde{c}'$ to (or removing $\tilde{a}'$ from) the state, but not by adding $\tilde{c}''$ (or removing $\tilde{a}''$) and, hence, $seq(\{\tilde{\gamma}, \tilde{\gamma}'\}[\tilde{c}, \tilde{c}''']) \not\subseteq seq(\{\tilde{\alpha}, \tilde{\gamma}'\}[\tilde{a}, \tilde{c}'''])$.

- The dynamic part of τ contains tokens not available in σ, but no message descriptor depends on them. A message descriptor in $\rho = \{\tilde{\gamma}'\}[\tilde{c}']$ can depend on the additional state descriptors such that $seq(\{\tilde{\gamma}, \tilde{\gamma}'\}[\tilde{c}, \tilde{c}']) \not\subseteq seq(\{\tilde{\alpha}, \tilde{\gamma}'\}[\tilde{a}, \tilde{c}'])$. □

Theorem 19 on page 77 and Theorem 20 on the page before show that type equivalence and subtyping for types not containing free type parameters are sound and complete with respect to equivalence and containment of sequence sets and the extensibility condition. All type parameters are eventually substituted by types without free type parameters.

5.7 Discussion

We have presented the process type concept. It is based on multi-sets of tokens that are used to express states. Before an object may send a message, the conditions on the multi-sets of tokens of the receiver must be fulfilled, and changes according to the type of the receiver are made. This type system is more expressive than TyCO, as shown in Section 5.5.2 on page 74. It is not able to express all types that were proposed by Ravara and Vasconcelos (see Section 3.5 on page 33), but they use a different notion of error. In their notion, errors do occur only when a message is not understood in all future contexts. We identify an error when a message is not understood in the context when it was received. So, Ravara and Vasconcelos fail to detect all message-not-understood errors, because a message not understood for the moment could be understood later.

The proposal from Colaço, Pantel and Sallé [CPS97] uses a notion of "orphan messages" which corresponds to messages that are not understood. They are able to compute different types in different contexts, but their work is not general enough to detect all orphan messages.

Kobayashi and Yonezawa [KY94] use reordering of messages, but cannot guarantee that all received messages are acceptable.

The type system from Najm and Nimour (Section 3.6.1 on page 37) is able to detect all messages that are not understood. Their types have two kinds of interfaces, public interfaces that are uniform, and private interfaces that are non-uniform. There may be an arbitrary number of references to public interfaces, but exactly one reference to private interfaces. Najm and Nimour allow the use of more than one interface for objects, but the restriction on either uniform or non-uniform interfaces is not very flexible. The interface types cannot be combined to one type. The concepts of type splitting and type combination cannot be supported because the notion of types is based on process algebra. They do not contain hidden information used for type splitting and type combination and their types are simpler than process types. But it is not possible to support any of the dynamic features that are presented for process types in the next section.

For the process types presented in this thesis, their expressiveness is similar to regular grammars. Equivalence and subtyping (which can be seen as containment of sequences) for regular process types have been presented in Section 5.6. It is easy to decide equivalence and containment in regular grammars. Process types, unlike other types must always be seen in the context of combination with other types. It is possible that a process type includes information which is not necessary at the moment, but when another type is combined with the type, the information becomes relevant. Therefore, two types are equivalent not only if their sequences are equal, but also if their combination with arbitrary types is equal. For example, the sequence of a type $\pi \stackrel{\text{def}}{=} \mathbf{\{}x()()[x][]\mathbf{\}}$ is empty, as well as the sequence of a type $\pi' \stackrel{\text{def}}{=} \mathbf{\{}x()()[y][]\mathbf{\}}$. But when a type $\pi'' \stackrel{\text{def}}{=} \mathbf{\{}z()()[][x]\mathbf{\}}$ is added, the sequences are different. Another example is a type $\sigma \stackrel{\text{def}}{=} \mathbf{\{}x()()[][z]\mathbf{\}}$ is not equivalent to $\tau \stackrel{\text{def}}{=} \mathbf{\{}x()()[][z,z]\mathbf{\}}$. There can be a $\rho \stackrel{\text{def}}{=} \mathbf{\{}y\mathbf{\}}[z]$ so that the $seq(\sigma|\rho) \neq seq(\tau|\rho)$.

The same problems arise when the subtype relation is examined. It is not enough, when the subtype understands all the sequences a supertype understands. Again we have to examine all possible combinations with other types. It is possible that redundant services become non-redundant after another type is combined to it.

Process types that have a similar expressiveness like non-regular grammars are presented in [Pun99]. For these types, equivalence and containment are not generally decidable. But, the notion of subtyping and equivalence is decidable, when the additional information hidden in process types is regarded.

Another version of process types may be used to prevent classes of deadlocks and livelocks [PP97, PP99b]. Deadlocks arise when all objects have no messages in the buffer that can be processed. Process types can be extended with obligatory messages that will be answered. Therefore, it is possible to use obligatory messages to detect unintended deadlocks.

Chapter 6

Higher Order Subtyping

6.1 Introduction

Subtyping is one of the best explored concepts in object-oriented programming. It is used as a basis for substitution. Recursive types allow the usage of types of the form *Self, Mytype* or *like Current*. Unfortunately, the subtyping relation does not hold between some recursively defined object types as reported in [CHOM89, DT88]. Binary methods use recursive types. For example:

$$\begin{aligned}\mathrm{Max} &\stackrel{\mathrm{def}}{=} \{\mathrm{max}()(\mathit{Self}\,[])[][]\}\\ \mathrm{MinMax} &\stackrel{\mathrm{def}}{=} \{\mathrm{max}()(\mathit{Self}\,[]), \mathrm{min}()(\mathit{Self}\,[])[][]\}\end{aligned}$$

where we assume the type *Self* to be defined as the static part of the current type. It is not possible to show the subtyping relation MinMax$\leq$Max although MinMax extends Max. The problem is that we require covariant (input) parameter types according to the definition of subtyping given by Cardelli in [Car88], while subtyping requires them to be contravariant. (This is expressed in the redundancy relation of process types which is used for parallel composition.) This problem is not a problem of the definition of subtyping [Cas95]. The next example shows that allowing subtyping for binary methods would be unsound:

$$\begin{aligned}\mathrm{Point} &\stackrel{\mathrm{def}}{=} \{\mathrm{getx}()(\mathbf{int}[])[][], \mathrm{gety}()(\mathbf{int}[])[][], \mathrm{equal}()(\mathit{Self}[],\mathbf{int}[])[][\mathrm{r,r}]\}[], \mathrm{ret}(\underline{u})(u)[r][]\\ \mathrm{CPoint} &\stackrel{\mathrm{def}}{=} \{\mathrm{getx}()(\mathbf{int}[])[][], \mathrm{gety}()(\mathbf{int}[])[][], \mathrm{getcol}()(\mathbf{int}[])[][],\\ &\qquad \mathrm{equal}()(\mathit{Self}[], \mathbf{int}[])[\mathrm{r,r}][], \mathrm{ret}(\underline{u})(u)[r][]\}[]\end{aligned}$$

The "equal" message inserts two "r" tokens in the state because the object then wants to receive two "ret" messages which remove the tokens.

The corresponding creator methods are shown in Figure 6.1 on the next page. We use an operation testing for the equality of objects presented in Section 6.3 on page 87 and assume that it compares integers, float, etc. correctly. The creator method "point" can be called with values x and y of type t. An object may receive messages that request the values of x and y. Asynchronous communication enforces that the sender of the message must send its address in the message in order to receive an answer. The answer is sent with the message "ret". A "point" object can also understand this message. The

$$
\begin{array}{ll}
\mathrm{point}(\underline{t})(\underline{x},\underline{y}) & \mathbf{\{}\mathrm{getx}()(\underline{\mathrm{r}})r\triangleleft\mathrm{ret}(t)(x).@\mathrm{point}(t)(x,y), \\
& \mathrm{gety}()(\underline{\mathrm{r}})r\triangleleft\mathrm{ret}(t)(y).@\mathrm{point}(t)(x,y), \\
& \mathrm{equal}()(\underline{o},\underline{r})\,o\triangleleft\mathrm{getx}()(\mathit{self}).\mathbf{\{}\mathrm{ret}(t')(x').x{=}x'\,?\;o\triangleleft\mathrm{gety}()(\mathit{self}). \\
& \quad\mathbf{\{}\mathrm{ret}(t')(y').y{=}y'\,?\;r\triangleleft\mathrm{true}()().@\mathrm{point}(t)(x,y)| \\
& \quad r\triangleleft\mathrm{false}()().@\mathrm{point}(t)(x,y)\mathbf{\}}|r\triangleleft\mathrm{false}()().@\mathrm{point}(t)(x,y)\mathbf{\}\}} \\
\mathrm{cpoint}(\underline{s})(\underline{x},\underline{y},\underline{c}) & \mathbf{\{}\mathrm{getx}()(\underline{\mathrm{r}})r\triangleleft\mathrm{ret}(s)(x).@\mathrm{point}(s)(x,y,c), \\
& \mathrm{gety}()(\underline{\mathrm{r}})r\triangleleft\mathrm{ret}(s)(y).@\mathrm{point}(s)(x,y,c), \\
& \mathrm{getcol}()(\underline{\mathrm{r}})r\triangleleft\mathrm{ret}(s)(c).@\mathrm{point}(s)(x,y,c), \\
& \mathrm{equal}()(\underline{o},\underline{r})\,o\triangleleft\mathrm{getx}()(\mathit{self}).\mathbf{\{}\mathrm{ret}(s)(x').x{=}x'? \\
& \quad o\triangleleft\mathrm{gety}(s)(\mathit{self}).\mathbf{\{}\mathrm{ret}(s')(y').y{=}y'\,? \\
& \quad o\triangleleft\mathrm{getcol}(s)(\mathit{self}).\mathbf{\{}\mathrm{ret}(s')(c').c{=}c'\,? \\
& \quad r\triangleleft\mathrm{true}()().@\mathrm{point}(s)(x,y)|r\triangleleft\mathrm{false}()().@\mathrm{point}(s)(x,y,c)\mathbf{\}} \\
& \quad r\triangleleft\mathrm{false}()().@\mathrm{point}(s)(x,y,c)|r\triangleleft\mathrm{false}()().@\mathrm{point}(s)(x,y,c)\mathbf{\}\}}
\end{array}
$$

Figure 6.1: Implementation of point and cpoint methods

“equal” message checks if the values are equal with the test for equality. If both values are equal, a “true” message is sent to the object that requested the test, else, “false” is sent.

The creator method “cpoint” behaves similarly, but contains a color-value which must also be compared. The following creator method illustrates how the subtyping relation of “Point” and “CPoint” is unsound:

$$\mathrm{breakit}()(p)(\underline{np})@\mathrm{point}(\mathbf{int})(1,1).p\triangleleft\mathrm{equal}(np,\mathit{self}).\ldots\mathbf{\{\}} : \mathrm{breakit}()(\mathrm{Point})[][]$$

When “breakit” is called with an actual parameter of type “Point”, everything is fine. However, when the actual parameter is of type “CPoint”, a run-time error would occur when $p\triangleleft\mathrm{equal}(np,\mathit{self})$ is evaluated. Since the type of p is “CPoint”, the code for “equal” in “cpoint” will be executed. When np receives the message “getc”, it can not understand this message. Therefore, in a sound type system, a call of “breakit” with an actual parameter of type “CPoint” must result in an error.

Languages like C++ [Str97, ES90], Java [AG96], Object Pascal [Tes85] and Modula-3 [CDG+88, Nel91] do not allow that the type of methods is changed upon subtyping. Eiffel [Mey92] allows argument types to be specialized. The above example would be correct in Eiffel. This covariant argument specialization would produce a run-time error when a “CPoint” is passed to “breakit”. There is a proposal to compensate this insecurity in the type system by a link-time data-flow analysis of the program (called system validity check) which would catch possible type errors. But then the subtype relation in Eiffel would have no clear meaning: Although “CPoint” would be a subtype of “Point”, “breakit” could not receive an argument of type “CPoint”.

Multi-methods have been proposed to solve the problem with binary methods [Cas95, Cha92]. They circumvent the problem by choosing an appropriate method implementation on behalf of the dynamic type of every argument of a message. However, this dynamic concept does not fit in our statically typed system and the subtyping relation must be specified explicitly.

Another solution to circumvent this problem is to use a relation between recursive object types called *matching*. Unlike subtyping, matching does not support substitution, but it supports inheritance of binary methods and parameterisation. The original interpretation of matching was in terms of F-bounded subtyping. F-bounded subtyping is undecidable [Pie94]. The common, more comfortable interpretation is now higher-order subtyping [AC96a].

There are some approaches to integrate matching in a programming language. The first one seems to be Emerald [BH91], which does not contain classes but support the use of bounded matching to restrict polymorphic functions. TOOPLE [Bru94] and PolyTOIL [BSvG95] are based on classes and supports subtyping as well as matching. Another language supporting subtyping and matching is Tool, described in [GM96].

PolyTOIL is very complex and has two hierarchies, one for subtyping and one for matching; the programmer can choose which one he uses, but the differences are not easy to understand for the programmer. The typing rules become quite unclear. For example, the distinction between matching and subtyping is unclear and given without proof in PolyTOIL.

The language LOOM [BPF97] emerged from experiences with PolyTOIL and does not provide any subtyping relation at all. It is completely based on the matching relation which is slightly different from the definitions we give later in the text. Their matching relation is based on invariant parameter types. No subtyping is supported. LOOM introduces hash-types. An object of a certain hash-type may be substituted by another object with a type that matches the hash-type. LOOM's matching relation is an extension relation: A type matches another type if it provides more methods than the other. Bruce et al. argue that this approach does not impact the expressiveness of the language significantly. However, our matching relation is still based on subtyping. But, we do not provide a subtyping hierarchy for the programmer.

6.2 Interfaces and Matching

The concept of interface is similar to the concept of protocol proposed by Abadi and Cardelli in [AC96a]. The syntax of *interfaces* ($\iota, \kappa, \lambda, \dots$) is given by:

$$
\begin{aligned}
\kappa &::= \chi \quad | \quad \psi \quad | \quad u \\
\chi &::= \sharp\boldsymbol{\{}\tilde{\alpha}\boldsymbol{\}}[\tilde{a}] \quad | \quad (\underline{t})\sharp\boldsymbol{\{}\tilde{\alpha}\boldsymbol{\}}[\tilde{a}] \quad | \quad \chi_1\|\chi_2 \quad | \quad \mathit{Self}[\tilde{a}] \\
\psi &::= \boldsymbol{\{}\tilde{\alpha}\boldsymbol{\}}[\tilde{a}] \quad | \quad (\underline{t})\boldsymbol{\{}\tilde{\alpha}\boldsymbol{\}}[\tilde{a}] \quad | \quad \psi_1\|\psi_2 \\
\alpha &::= x(\underline{\tilde{s}})(\tilde{\kappa})[\tilde{a}_1][\tilde{a}_2] \\
a &::= x \quad | \quad x^\infty
\end{aligned}
$$

An interface of the form $\boldsymbol{\{}\tilde{\alpha}\boldsymbol{\}}[\tilde{r}]$ consists of two parts, a static part $\boldsymbol{\{}\tilde{\alpha}\boldsymbol{\}}$ consisting of a set of message descriptors and a dynamic part, the state $[\tilde{r}]$ represented by a multi-set of tokens. An exact interface χ has an additional $\sharp$. Informally said, an when an object or parameter of an exact interface is expected, it cannot have a subinterface of this exact interface (See the definition of the interface equivalence relation below). Parameters (to be substituted by interfaces) and combinations of interfaces of the form $\chi_1 \| \chi_2$ or $\psi_q \| \psi_2$ are also (exact) interfaces. Recursion is allowed at the beginning by using a name ($\underline{t}$) as an abbreviation for the interface. The main difference to types is that an interface may also contain parameters of the distinguished self-reference s, where s is the name of the interface. This purely static reference to the type of the object does not contain any dynamic information, i.e., there is no multi-set of tokens in the type.[1]

Definition 49 (Substitution function) *We define a function $T(\kappa, \{\tilde{\alpha}\})$ which substitutes all occurrences of Self in the message descriptors $\{\alpha_1, \ldots, \alpha_n\}$ with an arbitrary type and associates them with new message descriptors $\{\alpha'_1, \ldots, \alpha'_n\}$, formally $\forall_{1 \le i \le n} \alpha_i \Rightarrow \alpha'_i \equiv \alpha_i[\kappa / Self]$. The function is also applicable to interfaces $T(\kappa, \lambda)$ $(\lambda \equiv \{\tilde{\alpha}\}[\tilde{a}])$; it returns a λ', formally $T(\kappa, \lambda) = T(\kappa, \tilde{\alpha})[\tilde{a}]$.*

An interface is deterministic if it is equivalent to an interface of the form $\{\tilde{\alpha}\}[\tilde{a}]$ such that all message descriptors $\tilde{\alpha}$ have pairwise different message selectors.

Two interfaces κ and λ are structurally equivalent, denoted by $\kappa \doteqdot \lambda$, if they can be made equal by renaming bound interface parameters (α-conversion) and repeatedly applying these rules to static and dynamic type parts:

$$\begin{array}{lll} [\tilde{a}, b, \tilde{c}] \doteqdot [\tilde{a}, \tilde{c}, b] & [x^\infty, x, \tilde{a}] \doteqdot [x^\infty, \tilde{a}] & [x^\infty, x^\infty, \tilde{a}] \doteqdot [x^\infty, \tilde{a}] \\ \{\tilde{\alpha}, \beta, \tilde{\gamma}\} \doteqdot \{\tilde{\alpha}, \tilde{\gamma}, \beta\} & \{\tilde{\alpha}, \beta, \beta\} \doteqdot \{\tilde{\alpha}, \beta\} & \end{array}$$

We define a set of relevant tokens for every name x annotated by $rel(x)$. It may contain tokens of the form $y^n (n > 0)$ and y^∞.

Definition 50 (Interface equivalence) *The interface equivalence relation $\Lambda \vdash \kappa \cong \lambda$ on types depending on a matching environment Λ is the least congruence closed under the rules in Figure 6.2 on the next page. We write $\kappa \cong \lambda$ instead of $\Lambda \vdash \kappa \cong \lambda$ if Λ is empty.*

Definition 51 (Redundancy relation) *The redundancy relation $\Lambda, X \vdash \alpha \gg \gamma$ on message descriptors, depending on a matching environment Λ and a set X of tokens, is defined by this rule:*

$$\Lambda, X \vdash x(\underline{\tilde{s}})(\tilde{\kappa})[\tilde{a}][\tilde{c}, \tilde{c}'] \gg x(\underline{\tilde{s}})(\tilde{\lambda})[\tilde{a}, \tilde{a}', \tilde{a}''][\tilde{a}'', \tilde{c}, \tilde{c}'']$$

where $\Lambda \vdash \tilde{\kappa} \lessdot \tilde{\lambda}$ (as defined below)and $\{\tilde{c}''\} \cap X = \emptyset$.

Interface combination is a commutative and associative operation with the interface $\{\}[]$ as neutral element. The message descriptor of a combination of two interfaces is the union of the two interfaces' message descriptors. The state of a combined interface is the multi-set-union of the two interface's states whereby all tokens in these states must be relevant. Irrelevant tokens can be removed.

[1] Parameters of this type are not allowed to occur in types because of the covariance-contravariance problem.

$$\Lambda \vdash \{\tilde{\alpha}\}[\tilde{a}] \simeq \{\tilde{\gamma}\}[\tilde{c}] \quad (\tilde{\alpha} \equiv \tilde{\gamma}) \tag{Struct$_\simeq$}$$
$$\Lambda \vdash \sharp\{\tilde{\alpha}\}[\tilde{a}] \simeq \sharp\{\tilde{\gamma}\}[\tilde{c}] \quad (\tilde{\alpha} \equiv \tilde{\gamma}) \tag{StructX$_\simeq$}$$
$$\Lambda \vdash (\kappa_1 \| \kappa_2) \| \kappa_3 \simeq \kappa_1 \| (\kappa_2 \| \kappa_3) \tag{Assoc$_\simeq$}$$
$$\Lambda \vdash \kappa \| \lambda \simeq \lambda \| \kappa \tag{Comm$_\simeq$}$$
$$\Lambda \vdash \boldsymbol{\{}\tilde{\alpha}\boldsymbol{\}}[\tilde{a}] \| \boldsymbol{\{}\tilde{\gamma}\boldsymbol{\}}[\tilde{c}] \simeq \boldsymbol{\{}\tilde{\alpha}, \tilde{\gamma}\boldsymbol{\}}[\tilde{a}, \tilde{c}] \quad ([\tilde{a}] \subseteq \mathit{rel}(\tilde{\alpha}); [\tilde{c}] \subseteq \mathit{rel}(\tilde{\gamma})) \tag{Par$_\simeq$}$$
$$\Lambda \vdash \sharp\boldsymbol{\{}\tilde{\alpha}\boldsymbol{\}}[\tilde{a}] \| \sharp\boldsymbol{\{}\tilde{\alpha}\boldsymbol{\}}[\tilde{c}] \simeq \sharp\boldsymbol{\{}\tilde{\alpha}\boldsymbol{\}}[\tilde{a}, \tilde{c}] \quad ([\tilde{a}] \subseteq \mathit{rel}(\tilde{\alpha}); [\tilde{c}] \subseteq \mathit{rel}(\tilde{\alpha})) \tag{ParX$_\simeq$}$$
$$\Lambda \vdash (\underline{t})\boldsymbol{\{}\tilde{\alpha}, \tilde{\gamma}\boldsymbol{\}}[\tilde{a}] \simeq (\underline{t})\boldsymbol{\{}\tilde{\alpha}\boldsymbol{\}}[\tilde{a}] \quad (\forall_{\gamma \in \{\tilde{\gamma}\}} \exists_{\alpha \in \{\tilde{\alpha}\}} \Lambda, \mathit{rel}(\tilde{\alpha}) \vdash \alpha \gg \gamma) \tag{Red$_\simeq$}$$
$$\Lambda \vdash (\underline{t})\boldsymbol{\{}\tilde{\alpha}\boldsymbol{\}}[\tilde{a}, b] \simeq (\underline{t})\boldsymbol{\{}\tilde{\alpha}\boldsymbol{\}}[\tilde{a}] \quad (b \notin \mathit{rel}(\tilde{\alpha})) \tag{NonRel$_\simeq$}$$
$$\Lambda \vdash \kappa \| \{\epsilon\} \simeq \kappa \tag{Emtpy$_\simeq$}$$
$$\Lambda \vdash (\underline{t})\boldsymbol{\{}\tilde{\alpha}\boldsymbol{\}}[\tilde{a}, x^\infty] \simeq (\underline{t})\boldsymbol{\{}\tilde{\gamma}\boldsymbol{\}}[\tilde{a}] \quad (\{\tilde{\gamma}\} = \mathit{del}_x(\tilde{\alpha})) \tag{Avail$_\simeq$}$$
$$\Lambda \vdash (\underline{t})\boldsymbol{\{}\tilde{\alpha}\boldsymbol{\}}[\tilde{a}] \simeq \boldsymbol{\{}\tilde{\alpha}[(\underline{t})\boldsymbol{\{}\tilde{\alpha}\boldsymbol{\}}[\tilde{a}]/t]\boldsymbol{\}}[\tilde{a}] \tag{Rec$_\simeq$}$$

Figure 6.2: Interface Equivalence Relation

The decidability of interface equivalence shall be shown by bringing the interfaces in *normal form*. The strong normalization procedure repeatedly applies the following rewrite rules

$$m + n + n' \to m + n'' \ (n'' = n + n') \tag{Add$_\to$}$$
$$[\tilde{r}, x^0] \to [\tilde{r}] \tag{Null$_\to$}$$
$$[\tilde{r}, x^m, x^n] \to [\tilde{r}, x^m + n] \tag{Sum$_\to$}$$
$$[\tilde{r}, x^\infty, x^n] \to [\tilde{r}, x^\infty] \tag{Inf1$_\to$}$$
$$[\tilde{r}, x^\infty, x^\infty] \to [\tilde{r}, x^\infty] \tag{Inf2$_\to$}$$
$$\Lambda \vdash \boldsymbol{\{}\tilde{\alpha}\boldsymbol{\}}[\tilde{a}] \| \boldsymbol{\{}\tilde{\gamma}\boldsymbol{\}}[\tilde{c}] \to \boldsymbol{\{}\tilde{\alpha}, \tilde{\gamma}\boldsymbol{\}}[\tilde{a}, \tilde{c}] \quad ([\tilde{a}] \subseteq \mathit{rel}(\tilde{\alpha}); [\tilde{c}] \subseteq \mathit{rel}(\tilde{\gamma})) \tag{Comb$_\to$}$$
$$\Lambda \vdash \sharp\boldsymbol{\{}\tilde{\alpha}\boldsymbol{\}}[\tilde{a}] \| \sharp\boldsymbol{\{}\tilde{\alpha}\boldsymbol{\}}[\tilde{c}] \to \sharp\boldsymbol{\{}\tilde{\alpha}\boldsymbol{\}}[\tilde{a}, \tilde{c}] \quad ([\tilde{a}] \subseteq \mathit{rel}(\tilde{\alpha}); [\tilde{c}] \subseteq \mathit{rel}(\tilde{\alpha})) \tag{Exact$_\to$}$$
$$\Lambda \vdash \boldsymbol{\{}\tilde{\alpha}\boldsymbol{\}}[\tilde{a}, \tilde{c}] \to \boldsymbol{\{}\tilde{\alpha}\boldsymbol{\}}[\tilde{a}] \quad (\tilde{c} \notin \mathit{rel}(\tilde{\alpha})) \tag{Red$_\to$}$$
$$\Lambda \vdash \kappa \| \{\epsilon\} \to \kappa \tag{Zero$_\to$}$$

and the equivalences for α-conversion, associativity and commutativity of $\|$, reordering of tokens in multi-sets until no further application of the rewrite rules is possible. The interface κ' resulting from applying the normalisation procedure to an interface κ is in normal form. Obviously, $\kappa \simeq \kappa'$ holds, because the rewrite rules are just directed version of $\simeq$ and the equivalence for multi-sets of tokens. If the rewrite rules and equivalences of the normalization procedure are applied in different orderings, the results are equal up to the above list of equivalences. Two interfaces are interface equivalent if and only if the corresponding interfaces in normal form are equal up to these equivalences. Equivalence of interfaces in normal form is decidable. It is possible to find a rating of all interface expressions such that the rating of an interface which matches the left hand of a rewrite rule is strictly larger than the rating of the corresponding interface at the right hand side. Thus and because the infimum in (comb$_\to$) is explicitly stated, the normalization procedure always terminates and $\kappa \simeq \lambda$ is decidable for interfaces κ and λ.

$$\Lambda \cup \{s \leqslant t\} \vdash s \leqslant t \qquad (\textsc{Assumption}_{\leqslant})$$

$$\Lambda \cup \vdash \mathit{Self}[\tilde{a}, \tilde{c}] \leqslant \mathit{Self}[\tilde{a}] \qquad (\textsc{Self}_{\leqslant})$$

$$\frac{\Lambda \vdash \kappa \simeq \lambda \| \iota}{\Lambda \vdash \kappa \leqslant \lambda} \qquad (\textsc{Match}_{\leqslant})$$

$$\frac{\Lambda \vdash \kappa \leqslant \lambda \quad \Lambda \vdash \kappa' \leqslant \lambda'}{\Lambda \vdash \kappa \| \kappa' \leqslant \lambda \| \lambda'} \qquad (\textsc{Par}_{\leqslant})$$

$$\frac{\Lambda \cup \{s \leqslant t\} \vdash \boldsymbol{\{}\tilde{\alpha}\boldsymbol{\}}[\tilde{a}] \leqslant \boldsymbol{\{}\tilde{\gamma}\boldsymbol{\}}[\tilde{c}]}{\Lambda \vdash (\underline{s})\boldsymbol{\{}\tilde{\alpha}\boldsymbol{\}}[\tilde{a}] \leqslant (\underline{t})\boldsymbol{\{}\tilde{\gamma}\boldsymbol{\}}[\tilde{c}]} \; (s \notin \mathit{fn}(\tilde{\gamma}, \Lambda);\; t \notin \mathit{fn}(\tilde{\alpha}, \Lambda)) \qquad (\textsc{Rec}_{\leqslant})$$

Figure 6.3: Matching

The matching relation on interfaces with dynamic information has to ensure that every message an interface understands is also understood by its subinterfaces. We also have to ensure that an exact interface cannot be replaced by another interface. In other words, an exact interface does not have any subinterfaces but itself. The matching relation is similar to the matching concept of Abadi and Cardelli [AC96a] by allowing parameters to be in the matching relation, too.[2]

Definition 52 (Matching) *The matching relation $\Lambda \vdash \kappa \leqslant \lambda$ on types, depending on a matching environment Λ, is the least reflexive and transitive relation closed under the rules in Figure 6.3. We write $\kappa \leqslant \lambda$ instead of $\Lambda \vdash \kappa \leqslant \lambda$ if Λ is empty.*

Theorem 21 $\kappa \leqslant \lambda \Leftrightarrow \forall \iota : T(\iota, \kappa) \leq T(\iota, \lambda)$

We let $\kappa \doteqdot \boldsymbol{\{}\tilde{\alpha}\boldsymbol{\}}[\tilde{a}]$ and $\lambda \doteqdot \boldsymbol{\{}\tilde{\gamma}\boldsymbol{\}}[\tilde{c}]$. Case a) $\kappa \leqslant \lambda \Rightarrow \forall \iota : T(\iota, \kappa) \leq T(\iota, \lambda)$. The basic rule for subtyping is Rule ($\textsc{Match}_{\leqslant}$). It states $\kappa \leqslant \lambda \implies \exists_{\iota} | \kappa \simeq \lambda \| \iota$. The interface equivalence rules concerning parallel composition state that ι may contain additional methods (Rule ($\textsc{Par}_{\leqslant}$)) or tokens (rules $\textsc{Par}_{\simeq}$ and $\textsc{Parx}_{\simeq}$). The additional methods may be removed using Rule ($\textsc{Nonrel}_{\simeq}$), additional tokens may be removed using Rule ($\textsc{Red}_{\simeq}$). This rule is also used to for matching according to Rule ($\textsc{Self}_{\leqslant}$).

Now let us take a closer look on the interfaces: If we apply the function T on κ and λ, we get $\kappa' \doteqdot \boldsymbol{\{}\tilde{\alpha}[\iota/\mathit{Self}\,]\boldsymbol{\}}[\tilde{a}]$ and $\lambda' \doteqdot \boldsymbol{\{}\tilde{\beta}[\iota/\mathit{Self}\,]\boldsymbol{\}}[\tilde{c}]$. If interface κ contains only additional tokens or methods and no parameters like *Self*, the subtyping relation holds because the rules for subtyping are equivalent to matching. If interface κ contains *Self* as a parameter, Rule ($\textsc{Red}_{\simeq}$) states that the parameters have to be supertypes (rule ($\mathrm{red}_{\cong}$)). This is true because $\iota \leq \iota$.
Case b) $\kappa \leqslant \lambda \Leftarrow \forall \iota : T(\iota, \kappa) \leq T(\iota, \lambda)$ is true because a type can be translated to an interface that contains no references to *Self*, and therefore, all the rules for matching hold.

[2] A different solution based on record extension was chosen in LOOM [BPF97].

$$P ::= a\langle B/\tilde{\mu}\rangle \qquad \mu ::= l(\tilde{\kappa})(\tilde{u}) \qquad M ::= l(\underline{\tilde{s}})(\underline{\tilde{x}})B$$
$$B ::= \{\tilde{M}\} \quad | \quad a\triangleleft\mu.B \quad | \quad a = b?C|D \quad | \quad \kappa\leqslant\lambda?\underline{a}.C|D \quad | \quad (\underline{a})@\mu.B \quad | \quad @\mu$$

Figure 6.4: Syntax of Objects

6.2.1 Examples of Interfaces

The next example shows the static part of a simple heterogenous buffer interface:

$$\begin{aligned} \mathit{Buffer} &\stackrel{\text{def}}{=} \{\text{put}(\underline{t})(t)[\text{empty}][\text{full}], \text{get}()(\mathit{Back}[\text{one}])[\text{full}][\text{empty}]\} \\ \mathit{Back} &\stackrel{\text{def}}{=} \{\text{back}(\underline{t})(t)[\text{one}][]\} \end{aligned}$$

Reuse of binary methods can be treated by using the matching relation. For example, the corresponding interfaces to the types "Max" and "MinMax" that were presented in Section 6.1 on page 81 are presented below:

$$\begin{aligned} \text{Max} &\stackrel{\text{def}}{=} \{\max()(\mathit{Back}[\text{one}], \mathit{Self}\,)[][]\} \\ \text{MinMax} &\stackrel{\text{def}}{=} \{\max()(\mathit{Back}[\text{one}], \mathit{Self}\,), \min()(\mathit{Back}[\text{one}], \mathit{Self}\,)[][]\} \end{aligned}$$

We assume that objects of the form "Max" and "MinMax" have an integer value stored. This value was inserted when the creator method (not printed here) was called. The objects will return the higher, respectively the lower value to the object matching the interface "Back[one]". As previously stated, it is not possible to prove the subtyping relationship $\mathit{MinMax} \leq \mathit{Max}$ because the occurrence of *Self* is used in a covariant way, violating the contravariance condition on input parameters. But, the matching relation $\mathit{MinMax}[] \leqslant \mathit{Max}[]$ is correct.

6.3 The Calculus with Dynamic Comparisons

In the following section we will add a dynamic comparison [Pun98] of object identity and a dynamic test of the matching relation to the previously defined operations on the process type calculus. In Section 6.3.1, the syntax will briefly be presented. In Section 6.3.2 on page 89, the reduction semantics are given.

6.3.1 Syntax

The syntax of expressions in the proposed calculus is given in Figure 6.4. Most of the expressions were already presented in Figure 5.1 on page 65. So, only a short description of the known expressions shall be given. The semantics of these expressions are described informally. A formal representation is given in Section 6.3.2 on page 89.

An active object (or processes) (denoted by $P, Q, R, \dots$) specifies object behavior. The name (identifier) of an object is a, the current behavior is B and $\tilde{\mu}$ is a queue of received and not handled messages.

A message $(\mu, \nu, \xi, \dots)$ of the form $x(\tilde{\kappa})(\tilde{u})$ associates a message selector x with a (possibly empty) list of interfaces $\tilde{\kappa}$ and a (possibly empty) list of object identifiers or object parameters $\tilde{u}$ used as actual parameters. The interfaces $\tilde{\kappa}$ can be used in partially specifying the behavior of the objects with identifiers $\tilde{u}$. Dynamically passing interfaces as parameters provides a flexible kind of genericity as shown below by examples.

Methods $(M, N, \dots)$ of the form $x(\underline{\tilde{s}})(\underline{\tilde{v}})B$ can be used within selections or as creator methods. Creator methods are invoked by expressions of the form $(\underline{a})@\mu.B$, the become operation is $@\mu$. Messages can be sent using $a\triangleleft\mu.B$.

A behavior $a{=}b?C|D$ equals C if a and b are (substituted by) equal object identifiers; otherwise it equals D. If a and b are different, b shall not occur free in C; because the names are checked to be equal, a can be used instead of b in C. An expression $\kappa{\leqslant}\lambda?(\underline{a})C|D$ equals $C[\iota/a]$ if κ matches λ, where ι is the difference between κ and λ; otherwise the behavior equals D.

The following example shows the type and creator methods of a printer demon. It prints documents on a selected printer. If the printer is busy, the printer demon delays the jobs. These are the static parts of the interfaces:

$$\begin{aligned}
\mathit{Doc} &\stackrel{\text{def}}{=} \{\text{print}()(\mathit{Printer}[], \mathit{PrintD}[])[][],\\
&\qquad \text{send_page}()(\mathit{Printer}[\text{ready}])[\text{page}][]\}\\
\mathit{PrintD} &\stackrel{\text{def}}{=} \{\text{q_doc}(\underline{s})(s, \mathit{Printer}[])[][\text{waiting}],\\
&\qquad \text{ok}()(\mathit{Printer}[\text{ready}])[\text{waiting}][], \text{busy}()()[\text{waiting}][]\}\\
\mathit{Printer} &\stackrel{\text{def}}{=} \{\text{ready}()(\mathit{PrintD}[])[][],\\
&\qquad \text{print_page}()(\mathit{Doc}[\text{page}])[\text{ready}][\text{ready}],\\
&\qquad \text{end_print}()()[\text{ready}][]\}
\end{aligned}$$

A document (an instance of *Doc*[...]) contains some pages and accepts a message "print" to print the pages to a certain printer via a certain printer demon. When requested by "send_page" the document sends a page to a printer.

The printer demon (an instance of *PrintD*[...]) tries to print a document immediately. If this is not possible (information is passed using the messages "ok" and "busy"), the printer demon postpones the message by sending it to itself, appending it to its message queue.

A client can ask a printer (an instance of *Printer*[...]) by sending the message "ready" if the printer is able to print a document. A page is printed by sending "print_page". The end of the printing job is indicated by the sending "end_print".

These are the creator methods:

$$\begin{array}{rl}
\text{doc}(\underline{s})(\underline{\mathit{self}}) & \{\ \text{print}()(\underline{u}, \underline{v})\ v\triangleleft\text{q_doc}(s)(\mathit{self}, u).@\text{doc}(s)(\mathit{self}),\\
& \ \ \text{send_page}()(\underline{v})\dots @\text{doc}(s)(\mathit{self})\}\\
\text{queue}()(\underline{\mathit{self}}) & \{\ \text{q_doc}(\underline{s})(\underline{u}, \underline{v})\ v\triangleleft\text{ready}()(\mathit{self}).@\text{wait}()(\mathit{self}, u, v)\}\\
\text{wait}(\underline{s})(\underline{\mathit{self}, u, v}) & \{\ \text{ok}()(w)@\text{print}(s)(\mathit{self}, u, w),\\
& \ \ \text{busy}()()\mathit{self}\triangleleft\text{q_doc}(s)(u, v).@\text{queue}()(\mathit{self}),\\
& \ \ \text{q_doc}(\underline{s'})(\underline{u'}, \underline{v'})\mathit{self}\triangleleft\text{q_doc}(s')(u', v').@\text{wait}(s)(\mathit{self}, u, v)\}\\
\text{print}(\underline{s})(\underline{\mathit{self}, u, v}) & \ \ s{\leq}\mathit{Doc}[\text{page}]?(\underline{s'})v\triangleleft\text{send_page}()(u).@\text{print}(s')(\mathit{self}, u, v)\ |\\
& \ \ v\triangleleft\text{end_print}()().@\text{queue}()(\mathit{self})\\
\text{prnt_ready}()(\underline{\mathit{self}}) & \{\ \text{ready}()(\underline{u})u\triangleleft\text{ok}()(\mathit{self})@\text{prnt_busy}()(\mathit{self})\}\\
\text{prnt_busy}()(\underline{\mathit{self}}) & \{\ \text{ready}()(\underline{u})u\triangleleft\text{busy}()().@\text{prnt_busy}()(\mathit{self}), \text{print_page}()(\underline{v})\dots,\\
& \ \ \text{end_print}()()@\text{prnt_ready}()(\mathit{self})\}
\end{array}$$

An instance of *Doc*[] is initialised by invoking the creator method "doc". The interface parameter $\underline{s}$ is to be substituted by the interface of the instance (for example, *Doc*[page^{15}] for a document with 15 pages). When the document receives "print", it sends its interface, a self reference and the printer's object identifier to the printer demon. Single pages are sent to a printer when the message "send_page" is received. A token "page" is removed from the dynamic part of the interface; the remaining number is equal to the actual number of pages that remain to be printed.

A printer demon of interface *PrintD*[] is created by invoking "queue". The only acceptable message is "q_doc" with interface information (of a document) and identifiers of a document and printer as actual parameters. It is checked whether the printer currently can print documents, using the token "waiting" to receive an answer. Then, the printer demon changes its behavior to "wait" (with the parameters $\underline{s}$ and $\underline{u}$ specifying the interface and identifier of a document and $\underline{v}$ the identifier of the printer) to receive the answer from the printer. If the printer returns "ok", the printer demon changes its behavior to "print". Otherwise it appends the message "q_doc" to the end of its own message queue and becomes "queue" again. If a "q_doc" message is received, this message is also appended to the end of the queue. The behavior "print" (with the same parameters as "wait") invokes itself recursively as long as there are pages in the document to be printed by sending "send_page". If no page is left, the printer demon's behavior becomes "queue" again.

If the printer receives "ready" while behaving as "prnt_ready", it returns "ok" and behaves as "prnt_busy". In this behavior, every "ready" message is answered with "busy". The printer accepts "print_page" messages and waits until the guaranteed "end_print" message is received. Then the printer behaves as "prnt_ready" again.

6.3.2 Reduction Semantics

The reduction semantics are an extension of the reduction semantics described in Section 4.3 on page 46, the a-execution relation is an extension of the rules given in Definition 42.

Definition 53 (a-execution relation) *The a-execution relation $\eta_{\{\tilde{M}\}} \stackrel{a}{\to} \theta_{\{\tilde{M}\}}$ on system configurations is the least closure of these rules given in Figure 6.5 on the next page.*

The execution relation $\eta_{\{\tilde{M}\}} \stackrel{*}{\to} \theta_{\{\tilde{M}\}}$ is the least reflexive and transitive closure of all a-execution relations $\eta_{\{\tilde{M}\}} \stackrel{a}{\to} \theta_{\{\tilde{M}\}}$ with $a \in \mathbb{X}$.

Rule (swap$_\to$) swaps the object identifiers or parameters to be compared and ensures that only the first of these two names can occur free in B. Rules (equal$_\to$) and (match$_\to$) remove conditionals that evaluate to true as soon as the results of the comparisons are known. Rule (not equal$_\to$) is applicable when both names to be compared are (or have been substituted by) object identifiers, and (not match$_\to$) when the subtype condition is not satisfied for all possible substitutions of types for type parameters in the compared types. Rules (call$_\to$), (select$_\to$) and (self send$_\to$) were given in Definition 42 on page 47.

$$\frac{\tilde{P} \equiv \tilde{P}' \quad \eta_{\{\tilde{M}\}} \cup \{\tilde{P}'\} \xrightarrow{a} \eta_{\{\tilde{M}\}} \cup \{\tilde{Q}'\} \quad \tilde{Q}' \equiv \tilde{Q}}{\eta_{\{\tilde{M}\}} \cup \{\tilde{P}\} \xrightarrow{a} \eta_{\{\tilde{M}\}} \cup \{\tilde{Q}\}} \ (a\langle B/\tilde{\mu}\rangle \in \eta) \qquad (\text{CONGR}_\rightarrow)$$

$$\eta_{\{\tilde{M}\}} \cup @l(\tilde{x}) \xrightarrow{a} \eta_{\{\tilde{M}\}} \cup B \quad (l(\underline{\tilde{x}})B \in \{\tilde{M}\}) \qquad (\text{CALL}_\rightarrow)$$

$$\eta_{\{\tilde{M}\}} \cup a\langle \boldsymbol{\{} l(\underline{\tilde{x}})B, \tilde{N} \boldsymbol{\}} / l(\tilde{x}), \tilde{\mu}\rangle \xrightarrow{a} \eta_{\{\tilde{M}\}} \cup a\langle B/\tilde{\mu}\rangle \qquad (\text{SELECT}_\rightarrow)$$

$$\eta_{\{\tilde{M}\}} \cup a\langle a \triangleleft \xi . B/\tilde{\mu}\rangle \xrightarrow{a} \eta_{\{\tilde{M}\}} \cup a\langle B/\tilde{\mu}, \xi\rangle \qquad (\text{SELF SEND}_\rightarrow)$$

$$\eta_{\{\tilde{M}\}} \cup \{a\langle b \triangleleft \xi . B/\tilde{\mu}\rangle, b\langle C/\tilde{\nu}\rangle\} \xrightarrow{a} \eta_{\{\tilde{M}\}} \cup \{a\langle B/\tilde{\mu}\rangle, b\langle C/\tilde{\nu}, \xi\rangle\} \qquad (\text{SEND}_\rightarrow)$$

$$\eta_{\{\tilde{M}\}} \cup \{a\langle (\underline{x}) @ \xi . B/\tilde{\mu}\rangle\} \xrightarrow{a} \eta_{\{\tilde{M}\}} \cup \{a\langle B[b/x]/\tilde{\mu}\rangle, b\langle @\xi[b/x]/\rangle\} \quad (b\langle C/\tilde{\mu'}\rangle \notin \eta) \qquad (\text{NEW}_\rightarrow)$$

$$\eta_{\{\tilde{M}\}} \cup a{=}b?B|C \xrightarrow{a} \eta_{\{\tilde{M}\}} \cup a{=}b?B[b/a]|C \qquad (\text{SWAP}_\rightarrow)$$

$$\eta_{\{\tilde{M}\}} \cup a{=}a?B|C \xrightarrow{a} \eta_{\{\tilde{M}\}} \cup B \qquad (\text{EQUAL}_\rightarrow)$$

$$\eta_{\{\tilde{M}\}} \cup a{=}b?B|C \xrightarrow{a} \eta_{\{\tilde{M}\}} \cup C \quad (a \neq b) \qquad (\text{NOT EQUAL}_\rightarrow)$$

$$\eta_{\{\tilde{M}\}} \cup \kappa \| \iota {\leqslant} \lambda ? (\underline{t}) B|C \xrightarrow{a} \eta_{\{\tilde{M}\}} \cup B[\kappa/t] \qquad (\text{MATCH}_\rightarrow)$$

$$\eta_{\{\tilde{M}\}} \cup \kappa {\leqslant} \lambda ? (\underline{t}) B|C \xrightarrow{a} \eta_{\{\tilde{M}\}} \cup C \quad (\nexists_{\tilde{\iota},\tilde{s}}\, \kappa[\tilde{\iota}/\tilde{s}] {\leqslant} \lambda[\tilde{\iota}/\tilde{s}]) \qquad (\text{NOT MATCH}_\rightarrow)$$

Figure 6.5: a-execution relation

6.4 Checking

A system configuration η is consistent if, for each object u in η, there is an execution step $\eta \xrightarrow{a} \eta'$, except if the object is waiting for a message to be received; an object with identifier u waits for a message if an expression of the form $u\langle B/\rangle$ is in η, i.e., the queue of received messages is empty. Other reasons for an object to be non-executable are regarded as errors. An error has occurred if

- η contains an object $u\langle B/\nu, \tilde{\mu}\rangle$ and (sel) is not applicable because no guarded process in $\boldsymbol{\{}\tilde{B}\boldsymbol{\}}$ can deal with ν;
- η contains an object $u\langle v \triangleleft \nu . B/\tilde{\mu}\rangle$ but E contains no expression $v\langle C/\rangle$, i.e., the referenced object with identifier $v \in \eta$ does not exist, or v is not an object identifier;
- η contains expressions like $u\langle v{=}w?B|C/\tilde{\mu}\rangle$ or $u\langle v {\leqslant} \kappa/\tilde{\mu}\rangle$, but one of v, w or κ is not a name or interface, respectively.

Furthermore, consistency of a system configuration η shall imply that each system configuration η' with $\eta \xrightarrow{*} \eta'$ also is consistent. Therefore the following definition of consistency of initial configurations is used:

Definition 54 (Consistency) *An initial configuration η is consistent if for each system configuration η' with $\eta \xrightarrow{*} \eta'$, where η' contains expressions of the form $u\langle \boldsymbol{\{}\tilde{M}\boldsymbol{\}}/\nu, \tilde{\mu}\rangle$ or $u\langle B/\mu\rangle$ with B not of the form $\boldsymbol{\{}\tilde{M}\boldsymbol{\}}$, there exists a system configuration η'' with $\eta' \xrightarrow{a} \eta''$.*

We propose a set of checking rules in Section 6.4.1 and show that the rules are consistent in Section 6.4.2 on page 93.

6.4.1 Checking Rules

An expression of the form $\Delta \vdash \langle B{:}\kappa \rangle$ means that the behavior B is compatible with the behavior partially specified by interface κ, depending on an environment Δ. The environment contains typings of the forms $v{:}\kappa$ (v is an object parameter of interface κ), and $x(\underline{\tilde{s}})(\tilde{\kappa}){:}\lambda$ (the program contains a creator method of signature $x(\underline{\tilde{s}})(\tilde{\kappa})$, the behavior of the creator method is partially specified by λ, the type parameters s may occur in λ). The static interface of the self-reference is "*Self*".

An environment of the form $\Delta[\tilde{v}{:}\tilde{e}]$ is constructed from Δ by adding the typings $\tilde{v}{:}\tilde{e}$. The names $\tilde{v}$ must not occur in Δ to the left of ":"; the names to the left of ":" of all typings in an environment are always pairwise different.

An expression of the form $\Delta \vdash \langle \tilde{e}, P{:}\kappa \rangle$ specifies that the additional checks (possibly causing modifications of Δ) expressed by $\tilde{e}_1$ must be performed before checking $P{:}\kappa$. Figure 6.6 on the next page lists the checking rules. The interface associated with a self-reference and the interface partially specifying a behavior B may be changed simultaneously in the same way (Rule (SELF on the following page)). If an interface can be split off (at a parallel composition) from the self reference's interface, messages specified by this interface can no longer be sent through the self-reference. Hence, since objects behaving according to B no longer need to be able to accept these messages, and the same type can be split off from the object's type. When there are exact interfaces, this operation cannot be applied. On the other side, if type information is added to the interface partially specifying B, an object of this interface can deal with all additional messages specified by this type.

Because we check for each process a client and a server view, we have different rules for replacing interfaces of references (Rule (SUB CLIENT)) and objects (Rule (SUB SERVER))with corresponding subinterfaces.

Rule (PARAM) deals with checks of the form $v{:}\kappa'$. An object identifier or object parameter v is used as argument, introducing a new alias of v. The corresponding parameter interface is κ. The rule requires $v{:}\kappa$ where $\kappa \trianglerighteq \kappa' \| \kappa''$ to be in the environment; v's mark is split into κ' and κ''; the new interface of v in the environment becomes κ''. To ensure that an exact interface cannot be split, we need the side condition using the equivalence relation.

Rule (SELECT) ensures in the side condition that there is at least one method for each message that may be received. All methods that can be activated are checked, this is done by checking the parameters and the behaviors for the methods in *act*.

When a message is sent to an object v, it must be ensured that the object understands that message, i.e. the message signature is enabled in the current interface of v. Then it must be checked if the actual parameters correspond to the formal parameters.

If the interface of the receiver is not exact, there must not be any occurrences of $\mathit{Self}[\tilde{b}]$ in the receiver's formal parameter list (Rule (SEND)). If the receiver's interface is exact, $\mathit{Self}[\tilde{b}]$ may be in the receiver's parameter list (Rule (SEND). The reason for this restriction is that when the receiver's interface is not exact, $\mathit{Self}[\tilde{b}]$ is also not exact.

Therefore, it is not possible to check at compile-time, if the actual parameter has the correct interface. This solution is also used in LOOM [BPF97].

Object identity comparisons are dealt with in Rule (Equal): The environment must associate marks with the compared names. Behavior B is executed if the objects are identical (one name is replaced by the other and the marks are combined); else behavior C is executed. Interface combination requires that the interfaces contain no exact interfaces. But this condition is always fulfilled. The interface of references may be split by send and call rules. Both rules use Rule (Param), which ensures that the interfaces are split correctly and therefore are combinable again.

Interface comparisons are handled by Rule (Sub). The interface at the left of the $\leqslant$ must be an interface parameter not occurring in the superinterface at the right. If the subinterface relation holds, this interface parameter is substituted by the combination of the given superinterface and the new interface parameter specifying the difference between the sub- and the superinterface.

The Rule (Call) checks if there is a creator method of corresponding signature and behavior. The actual object parameters are checked if they have appropriate interfaces.

$$\frac{\Delta\vdash\langle B{:}\iota\rangle}{\Delta\vdash\langle B{:}\kappa\rangle}\,(\kappa\leqslant\iota) \qquad (\textsc{Sub Server})$$

$$\frac{\Delta\cup\{u{:}\iota\}\vdash\langle B{:}\lambda\rangle}{\Delta\cup\{u{:}\kappa\}\vdash\langle B{:}\lambda\rangle}\,(\kappa\leqslant\iota) \qquad (\textsc{Sub Client})$$

$$\frac{\Gamma\cup\{u{:}\kappa'\}\vdash\langle B{:}\kappa''\rangle}{\Gamma\cup\{u{:}\kappa\|\lambda\}\vdash\langle B{:}\iota\|\lambda\rangle}\,(\exists_{\kappa',\kappa'',\lambda'}|\kappa' \simeq \kappa\|\lambda' \vee \kappa'' \simeq \iota\|\lambda') \qquad (\textsc{Self})$$

$$\frac{\forall_{i=1..n}\,\Delta\cup\{\tilde{v}_i{:}\tilde{\kappa}_i\}\vdash\langle B_i{:}\boldsymbol{\{}\tilde{\alpha}\boldsymbol{\}}[\tilde{c}_i]\rangle}{\Delta\vdash\langle\boldsymbol{\{}x_1(\underline{\tilde{s}}_1)(\underline{\tilde{v}}_1)B_1,\ldots,x_n(\underline{\tilde{s}}_n)(\underline{\tilde{v}}_n)B_n\boldsymbol{\}}{:}\boldsymbol{\{}\tilde{\alpha}\boldsymbol{\}}[\tilde{a}]\rangle}\quad(1) \qquad (\textsc{Select})$$

$$\frac{\Delta\cup\{v{:}\boldsymbol{\{}x(\underline{\tilde{s}})(\tilde{\kappa})[\tilde{a}][\tilde{c}]\boldsymbol{\}}[\tilde{c}]\|\lambda\}\vdash\langle\tilde{w}{:}\tilde{\kappa}[\tilde{\iota}/\tilde{s}],B{:}\kappa'\rangle}{\Delta\cup\{v{:}\boldsymbol{\{}x(\underline{\tilde{s}})(\tilde{\kappa})[\tilde{a}][\tilde{c}]\boldsymbol{\}}[\tilde{a}]\|\lambda\}\vdash\langle v{\triangleleft}x(\tilde{\iota})(\tilde{w}).B{:}\kappa'\rangle}\,(\nexists_{\kappa''\in\tilde{\kappa},\tilde{b}}|\kappa''=\mathit{Self}[\tilde{b}]) \qquad (\textsc{Send})$$

$$\frac{\Delta\cup\{v{:}\sharp\boldsymbol{\{}x(\underline{\tilde{s}})(\tilde{\kappa})[\tilde{a}][\tilde{c}]\boldsymbol{\}}[\tilde{c}]\|\lambda\}\vdash\langle\tilde{w}{:}\tilde{\kappa}[\tilde{\iota}/\tilde{s}],B{:}\kappa'\rangle}{\Delta\cup\{v{:}\sharp\boldsymbol{\{}x(\underline{\tilde{s}})(\tilde{\kappa})[\tilde{a}][\tilde{c}]\boldsymbol{\}}[\tilde{a}]\|\lambda\}\vdash\langle v{\triangleleft}x(\tilde{\iota})(\tilde{w}).B{:}\kappa'\rangle} \qquad (\textsc{Sendx})$$

$$\frac{\Delta\cup\{v{:}\kappa\|\lambda\}\vdash\langle B{:}\iota\rangle\quad\Delta\cup\{v{:}\kappa,w{:}\lambda\}\vdash\langle C{:}\iota\rangle}{\Delta\cup\{v{:}\kappa,w{:}\lambda\}\vdash\langle v{=}w?B|C{:}\iota\rangle} \qquad (\textsc{Equal})$$

$$\frac{\Delta\vdash\langle B[\kappa\|t/s]{:}\lambda\rangle\quad\Delta\vdash\langle C{:}\lambda\rangle}{\Delta\vdash\langle s{\leq}\kappa?(\underline{t})B|C{:}\lambda\rangle}\,(s{\notin}\mathit{fn}(\kappa)) \qquad (\textsc{Sub})$$

$$\frac{\Delta\cup\{x(\underline{\tilde{s}})(\tilde{\kappa}){:}\lambda\}\vdash\langle\tilde{w}{:}\tilde{\kappa}[\tilde{\iota}/\tilde{s}],\boldsymbol{\{\}}{:}\lambda'\rangle}{\Delta\cup\{x(\underline{\tilde{s}})(\tilde{\kappa}){:}\lambda\}\vdash\langle @x(\tilde{\iota})(\tilde{w}){:}\lambda[\tilde{\iota}/\tilde{s}]\|\lambda'\rangle} \qquad (\textsc{Call})$$

$$\frac{\Delta\cup\{v{:}\kappa''\}\vdash\langle\tilde{w}{:}\tilde{\lambda},B{:}\iota\rangle}{\Delta\cup\{v{:}\kappa\}\vdash\langle v{:}\kappa',\tilde{w}{:}\tilde{\lambda},B{:}\iota\rangle}\quad(\kappa \simeq \kappa'\|\kappa'') \qquad (\textsc{Param})$$

(1) $\boldsymbol{\{}\tilde{\alpha}\boldsymbol{\}}[\tilde{a}]$ deterministic; $\{x_1,\ldots,x_n\}=\{x \mid x(\tilde{s})(\tilde{\kappa})[\tilde{c}]{\in}\mathit{act}(\boldsymbol{\{}\tilde{\alpha}\boldsymbol{\}}[\tilde{a}])\}$;
$x_i(\tilde{s}_i)(\tilde{\kappa}_i)[\tilde{c}_i]{\in}\mathit{act}(\boldsymbol{\{}\tilde{\alpha}\boldsymbol{\}}[\tilde{a}])$;

Figure 6.6: Checking rules

6.4.2 Consistency of Initial Configurations

Definition 55 (Interface consistency) *A set $\{\tilde{M}\}$ of creator methods (considered to be a program) is type-consistent with respect to an environment Δ if*

- *each expression in Δ is of the form $x(\underline{\tilde{s}})(\tilde{\kappa}){:}\iota$, and for each such expression there is a creator method $x(\underline{\tilde{s}})(\tilde{v})B \in \{\tilde{M}\}$ with $|\tilde{\kappa}| = |\tilde{v}|$,*
- *for each $x(\underline{\tilde{s}})(\tilde{v})B{\in}\{\tilde{M}\}$ there is an expression $x(\underline{\tilde{s}})(\tilde{\kappa}){:}\iota{\in}\Delta$ such that $\Delta{\cup}\{\tilde{v}{:}\tilde{\kappa}\} \vdash \langle B{:}\iota\rangle$ as specified by the rules in Figure 6.6 on the page before.*

The first condition ensures that there are implementations of creator methods for all expressions in the environment. The second condition ensures that each creator method in a program behaves as partially specified by the expressions in the environment. The corresponding type checks can be performed separately for each creator method.

Theorem 22 *Let $\{\tilde{M}\}$ be a program consistent with respect to an environment Δ and $\{x\langle @y()()/\rangle\}_{\{\tilde{M}\}}$ a system configuration such that there is a $y(){:}\kappa \in \Gamma$. Then, for each system configuration $\eta_{\{\tilde{M}\}}$ with $\{x\langle @y()()/\rangle\}_{\{\tilde{M}\}} \overset{*}{\to} \eta_{\{\tilde{M}\}}$ and each $z\langle B/\tilde{\mu}\rangle \in \eta_{\{\tilde{M}\}}$ with $|\tilde{\mu}| > 0$ or B not of the form $\mathbf{\{}\tilde{N}\mathbf{\}}$, there is a system configuration $\theta_{\{\tilde{M}\}}$ with $\eta_{\{\tilde{M}\}} \overset{z}{\to} \theta_{\{\tilde{M}\}}$.*

PROOF. The proof by case analysis is sketched. For each system configuration $\eta_{\{\tilde{M}\}}$ and each $z\langle B/\tilde{\mu}\rangle \in \eta_{\{\tilde{M}\}}$ with $|\tilde{\mu}| > 0$ or B not of the form $\mathbf{\{}\tilde{N}\mathbf{\}}$ it has to be shown that (case a) there exists a system configuration $\theta_{\{\tilde{M}\}}$ with $\eta_{\{\tilde{M}\}} \overset{z}{\to} \theta_{\{\tilde{M}\}}$ or (case b) $\{x\langle @y()()/\rangle\}_{\{\tilde{M}\}} \overset{*}{\to} \eta_{\{\tilde{M}\}}$ cannot hold under the conditions of the theorem.

Case 1. Let an object $z\langle \mathbf{\{}N\mathbf{\}}/x'(\tilde{\lambda})(\tilde{v}),\tilde{\mu}\rangle$ be in $\eta_{\{\tilde{M}\}}$. If $\{\tilde{N}\}$ contains a method $x'(\underline{\tilde{s}})(\underline{\tilde{w}})C$ with $|\tilde{\lambda}|{=}|\tilde{s}|$ and $|\tilde{v}|{=}|\tilde{w}|$, (case a) there is a $\theta_{\{\tilde{M}\}}$ with $\eta_{\{\tilde{M}\}} \overset{z}{\to} \theta_{\{\tilde{M}\}}$.

Otherwise let $\Delta \vdash \langle \mathbf{\{}\tilde{N}\mathbf{\}}{:}\mathbf{\{}\tilde{\alpha}\mathbf{\}}[\tilde{a}]\rangle$ for an environment Δ and interface $\mathbf{\{}\tilde{\alpha}\mathbf{\}}[\tilde{a}]$. The Rule (Select) requires that $\{\tilde{N}\}$ contains a method satisfying the condition of case a) for each expression in $act(\mathbf{\{}\tilde{\alpha}\mathbf{\}}[\tilde{a}])$. In case b) we assume that there is no expression $x'(\underline{\tilde{s}})(\tilde{\kappa})[\tilde{c}]$ with $|\tilde{\lambda}| = |\tilde{s}|$ and $|\tilde{v}| = |\tilde{\kappa}|$ in $act(\mathbf{\{}\tilde{\alpha}\mathbf{\}}[\tilde{a}])$. The interface $\mathbf{\{}\tilde{\alpha}\mathbf{\}}[\tilde{a}]$ is deterministic. An object must have sent $x'(\tilde{\lambda})(\tilde{v})$ to z. Rule (Select) requires $w{:}\iota \in \Delta'$, where w is the object parameter to be replaced by z, Δ' the environment of the sender, and ι a subinterface of an interface $\mathbf{\{}\tilde{\gamma}\mathbf{\}}[\tilde{a}']$ with $x'(\underline{\tilde{s}})(\tilde{\kappa})[\tilde{c}] \in act(\mathbf{\{}\tilde{\gamma}\mathbf{\}}[\tilde{a}'])$.

The expression $w{:}\iota$ is introduced into Δ' as parameter of a message. Let us ignore dynamic changes of types for the moment. An analysis of the rules in Figure 6.6 shows that in each case $\mathbf{\{}\tilde{\alpha}\mathbf{\}}[\tilde{a}] \le \iota'$, where ι' is constructed from ι by substituting interface parameters. When ignoring interface updating, the conditions of Definition 55 cannot be satisfied in case b) because it is impossible that superinterfaces of ι contain an active message descriptor, while types corresponding to $\mathbf{\{}\tilde{\alpha}\mathbf{\}}[\tilde{a}]$ do not.

Now let us consider dynamic changes of interfaces. Interface consistency is checked when all message queues are empty. Rule (Param) ensures that an interface is split into two parts along a parallel composition whenever an object identifier (or a corresponding formal parameter) is used as actual parameter and splitting is possible. One part of the interface remains with the formal parameter, the other part is associated with the actual parameter. The object of identifier z behaves as partially specified by $\mathbf{\{}\tilde{\alpha}\mathbf{\}}[\tilde{a}]$ with

$\{\tilde{\alpha}\}[\tilde{a}] \leq \iota_1\|\ldots\|\iota_n$, where $\iota_1,\ldots,\iota_n$ are the interfaces of all object parameters to be substituted by z (after substituting interface parameters). Since $\{\tilde{\alpha}\}[\tilde{a}]$ is deterministic, (send) changes the types $\iota_1,\ldots,\iota_n$ in the same way as (select) does for $\{\tilde{\alpha}\}[\tilde{a}]$. $\{\tilde{\alpha}\}[\tilde{a}'] \leq \iota_1'\|\ldots\|\iota_n'$ holds for $\{\tilde{\alpha}\}[\tilde{a}']$ and $\iota_1',\ldots,\iota_n'$ constructed by changing $\{\tilde{\alpha}\}[\tilde{r}]$ and $\iota_1,\ldots,\iota_n$, respectively. The same holds for rules (sendx) in combination with (select).

Hence, the conditions of Definition 55 cannot be satisfied in case b).

Case 2. Let $z\langle v\triangleleft\xi.C/\tilde{\mu}\rangle$ be in $\eta_{\{\tilde{M}\}}$. If v is a self-reference or an object of identifier v occurs in $\eta_{\{\tilde{M}\}}$, then (case a) there is a $\theta_{\{\tilde{M}\}}$ with $\eta_{\{\tilde{M}\}} \xrightarrow{z} \theta_{\{\tilde{M}\}}$.

Otherwise (case b) let$\Delta \vdash \langle v\triangleleft\xi.C{:}\iota\rangle$ for some Δ and ι. Rule (Select) requires $v{:}\lambda \in \Delta$. Such expressions are introduced into Δ for formal object parameters. The rules in Figure 6.6 on page 92 ensure that the environments contain expressions of the form $w{:}\kappa$ for all actual parameters w substituting such formal parameters. Thus, there exists an object for each $v{:}\lambda \in \Delta$, and the conditions of Definition 55 cannot be satisfied in case b).

Case 3. Let $z\langle @x'(\tilde{\lambda})(\tilde{v})/\tilde{\mu}\rangle$ be in $\eta_{\{\tilde{M}\}}$. If $\{\tilde{M}\}$ contains a creator method of the form $x'(\underline{\tilde{s}})(\underline{\tilde{w}})B$ with $|\tilde{\lambda}| = |\tilde{s}|$ and $|\tilde{v}| = |\tilde{w}|$, (case a) there is a $\theta_{\{\tilde{M}\}}$ with $\eta_{\{\tilde{M}\}} \xrightarrow{z} \theta_{\{\tilde{M}\}}$.

In case b) let $\Delta \vdash \langle @x'(\tilde{\lambda})(\tilde{v}){:}\iota\rangle$ or $\Delta \vdash \langle(\underline{v}')@x'(\tilde{\lambda})(\tilde{v}).C{:}\iota\rangle$ for some Δ and ι. Since the rule (call) requires that Δ contains an expression $x'(\underline{\tilde{s}})(\underline{\tilde{w}}){:}\kappa$, the conditions of Definition 55 cannot be satisfied in case b).

Case 4. For all other objects in $\eta_{\{\tilde{M}\}}$ there is always a $\theta_{\{\tilde{M}\}}$ with $\eta_{\{\tilde{M}\}} \xrightarrow{z} \theta_{\{\tilde{M}\}}$. $\square$

6.5 Discussion

The problems with the subtyping relation of some recursively defined objects [CHOM89] may be solved with matching [BCC+95]. The interpretation of matching as higher order subtyping [PS97] is the most preferable solution [AC96a].

Matching was integrated differently in a couple of sequential languages to test its interaction with subtyping, for example the functional object-oriented language TOOPLE [Bru94, BCK93] and its imperative relative PolyTOIL [BSvG95], both presented by Bruce et al. PolyTOIL supports a subtype and matching relation on object types. A later approach also integrating type quantification was made by Gawecki and Matthes[GM96]. Both approaches result in complex systems for type checking, using two relations that can hardly be distinguished by programmers.

Therefore, Bruce et al. decided to abandon the subtype relation and support only the matching relation in LOOM [BPF97]. The matching relation is different from the matching relation in this thesis and also from [AC96a]. It is based on record extension with invariant parameters. Bruce et al. argue that in their approach matching is able to replace subtyping.

There is a concurrent version of LOOM [Bro97], which extends LOOM with some primitives for synchronization. This approach uses pre- and post-conditions as synchronization constraints which results in problems with inheritance.

Our approach also does not support a subtyping hierarchy for the programmer but still uses subtyping to decide the matching relation. We use interfaces which can be mapped to types. The interfaces also have multi-sets of tokens and the matching relation also deals with token-sets. We have shown that it is possible to use the concepts of process types also for matching.

Part III

An Application: Process Types in CORBA

Chapter 7

Dynamic Type Checking

7.1 Introduction

In the previous part of our work, a powerful type concept was presented. Process types are designed to support static type checking. They are based on a newly designed calculus which is not easy to integrate in well known languages like C++, Ada, or Java because the types must be updated according to the execution path.

Distributed systems consist of several program parts written in different languages supporting different programming mechanisms and concepts. Middleware like CORBA allows distributed objects in such systems to communicate with each other in a uniform but flexible way. The interfaces of the distributed objects are described in a special language, the interface definition language (IDL). The interface details the services supported and their signature: a list of parameters with their IDL type and parameter-passing mode, the IDL type of the value, the exceptions that may possibly be raised during the processing of the service. Each object that needs a service from another object must access the service using the object broker. But, except for the parameters of the object's service, there is no information about the expected use of the service.

In the following we will examine how to integrate concepts of process types in the CORBA interface definition language [PP98]. A server object written in a specific language provides a set of operations for an arbitrary set of clients written in arbitrary languages. The only language that clients as well as the server understand is the interface definition language. A CORBA IDL-specification of a server-interface is compiled to a "skeleton" in the client's target language [Obj99] which is included when the client is compiled. The skeleton includes the stub-procedure calls to invoke the server's operations.

When we extend the CORBA-IDL to support process type information, static type checking becomes impossible because we cannot change the compilers of the target languages, and even if we did, we could not trace the execution paths effectively.

Additionally, middleware platforms should not change the languages its objects use. It is also desirable to determine messages that may not be understood before the client object sends them to the server. This goal can be reached by adding a checker at the CORBA-ORB which checks at run-time if "message-not-understood-errors" may occur.

This checker keeps track of the messages accepted by the server object. Instead of sending a non-acceptable message, a local exception is raised.

The rest of this chapter is structured the following way: Section 7.2 examines how process types can be integrated in the CORBA concept. Section 7.3 describes the control mechanisms the checker uses. Section 7.4 discusses various approaches to extend the expressiveness of the CORBA-IDL.

7.2 The CORBA-IDL with Process Types

The Common Object Request Broker Architecture (CORBA) is a standard proposed by the Object Management Group (OMG) in order to improve object interoperability between independently developed applications across heterogenous computer networks. This goal shall be achieved by using the object-based technology and its benefits like increased reusability, maintainability, interoperability and portability. Other goals are to standardise the best understood and best supported features of distributed objects systems, promote interoperability between heterogenous software systems, and integrate popular object-oriented and non-object-oriented programming languages such as C, C++, COBOL or Java.

Application objects are created by the software designers and the Object Request Broker (CORBA-ORB) provides the infrastructure for communication in a distributed object environment.

Application objects are often developed using object-oriented paradigms, but these "old" applications, tools, etc. can participate in the architecture using adapters or wrapper techniques.

The object model defined by CORBA has the following properties that must be compared to the requirements on the process type model given in Section 4.2 on page 45:

- **Object identity**: Each object has an identifier (its name) that provides a means to refer to the object and is immutable and persists as long as the object exists.

- **Client/Server relationship**: The distributed object system consists of a set of objects that interact by invoking services of another. One of the objects then acts as a client, requesting a service and the other acts as a server, waiting for requests. Normally an object acts both as a server and a client at different moments of its activity.

- **Access through references**: A client object can request a service from a server object only if it has a reference to that server. The invocation is using a strongly typed interface. This is the basic type of interaction, although CORBA supports also the dynamic construction of invocations, to allow contacting objects whose class is not known at compile time.

- **(Normally) synchronous invocation**: An invocation is a two-ways question and response interaction, where parameters are sent and results are returned. The call is blocking. Newer models for CORBA also allow asynchronous message sending (oneway) and there are proposals [OMG98b] for callback invocation (the answer is processed by a callback-object) and polling invocation (a polling object is used to retrieve the answer).

The first three properties described above are fulfilling the requirements of the process type model. The asynchronous invocation mechanisms are problematic because the (local) ordering of messages is not explicitly guaranteed by the CORBA-communication system and may depend on the implementation. Therefore we propose either to ensure this property by the communication system (for example by ensuring that the messages send back an acknowledgement when they are received) or to use synchronous communication only.

The ORB is the transport mechanism for the entire object communication and connects all instances with each other. This mechanism is independent of the location of the objects, the operating system and the programming languages in which the objects are coded. If an object (client) wants to make a request to an operation of another object (server), it has to contact the ORB. The client can use two techniques to make a service-request: Use the static client IDL stub or use the Dynamic Invocation Interface (DII). The current ORB core's job is only to create calls via stubs or provide information for the DII and then execute the DII invocation. We propose to extend the ORB's capabilities with coordination mechanisms. While the client accesses the ORB via stub procedure or DII, the Process-Type-Checker (PT-checker) checks if the call is a part of the allowed message sequence of the server object. The additional information about the server object, called the type mark, is stored at the client side because every client has its own view of the type mark of the server object.

The interfaces of CORBA Objects are specified using the OMG IDL [OMG98a]. These interfaces just provide information about the parameters of the operations the objects offer, but no information about the expected order of the operation calls. Some extensions of the CORBA-IDL syntax enable the CORBA-IDL to specify the expected sequence of messages according to the concept of process types. The PT-checker checks every time a service is called if the call is a part of the allowed message sequence. We extend the syntax to enable declaration and handling of tokens in interfaces:

interface ⟨name⟩:⟨base_interface⟩ **{**
  token ⟨token⟩ {, ⟨token⟩};
  ...
  initial ⟨token-list⟩;
}

The new keyword **token** defines token identifiers. The tokens defined in the interface can be included in the type mark. The keyword **initial** defines the type mark which is constructed after the object has been initialised. ⟨constant⟩, ⟨type⟩, ⟨exception⟩ and ⟨attribute⟩ remain the same, the operation specification is changed:

⟨new-op⟩::= [**oneway**] ⟨type⟩ ⟨identifier⟩ ([⟨new-param⟩ {, ⟨new-param⟩}])
    [**require** ⟨token-list⟩] [**ensure** ⟨token-list⟩] ⟨exceptions⟩ ⟨context⟩
⟨token-list⟩::= ⟨token⟩ [∗ ⟨positive-integer⟩] {, ⟨token⟩ [∗ ⟨positive-integer⟩]}
⟨new-param⟩::= [**in** | **out** | **inout**] ⟨type⟩ [< ⟨token-list⟩ >] ⟨identifier⟩

Preconditions of an operation are defined using the keyword **require**, postconditions using **ensure**. A token condition specified in the **require** or **ensure** clause can be either a simple token or a number of n tokens.

For example the previously presented buffer that can store arbitrary elements shall be specified. The condition on the message sequence can be specified with the following interface definition:

interface Buffer {
  token full, empty;
  void put (**in** T r) **require** empty **ensure** full;
  void get (**out** T r) **require** full **ensure** empty;
  void increment (**in unsigned int** n) **ensure** empty $*$ n;
}

Buffer describes a generic object type taking an arbitrary type expression as parameter. Memory for the elements must be allocated explicitly. Two tokens, "full" and "empty" and three operations are specified. The operation "put" inserts an element r of type T into an instance of Buffer. The **require**-clause assures as a precondition that there has to be at least one token "empty" in the multi-set representing the abstract state. When such a message is accepted, an "empty" token is removed from the multi-set and a "full" is added as specified in the **ensure**-clause (postcondition). In the same way, a message "put" changes the abstract state by replacing an empty buffer slot with a full one. The method "increment" increments the size of an instance of Buffer by n slots, it allocates the required amount of memory for n elements. The postcondition specifies that n slots of "empty" are added to the multi-set when "increment" is accepted. After the creation of the object, the type of the object is Buffer<>. The operation "increment" with a parameter n has to be called in order to initialise the object. The object's state becomes Buffer<empty$*$n>, there are n empty slots and the object will accept as may put messages as there are empty slots. If the object receives a put operation, the buffer's type changes to Buffer<full, empty$*$(n$-$1)>. After i ($i \leq n$) "put" calls, the Buffer's type is Buffer<full$*$i, empty$*$(n$-$i)>. If a "get" operation is called, the type is Buffer<full$*$(i$-$1), empty$*$(n$-$i+1)>.

The CORBA-IDL allows to specify one or more base-interfaces from which the new interface is derived. The subtyping rules follow the the principle of substitutability [LW93, LW94, WZ88] because every sub-interface understands the operations of all its super-interfaces. It is not possible to omit any operations. Our IDL extension should also support maximum flexibility based on the substitution principle. In order to do that, we have to extend the subtyping rules with rules for assertions.

S is a subtype of T if:

- S declares all operations declared by T with the same modifiers. The input and output parameters have to be invariant.

- The multi-set of tokens in S' require-clause has to be a subset of the multi-set of tokens in T's require-clause.

- The ensure-clause of S contains at least the same number of tokens occurring in the ensure-clause of T, except of tokens that are not declared by S.

It may seem surprising that a subtype does not have to declare all tokens declared by its supertypes. The reason is that tokens are used for the description of constraints on message sequences. A subtype is allowed to understand more message sequences than its supertypes. This can be achieved by omitting constraints. If constraints are removed, it is also possible to remove the tokens that are used to specify these constraints. For example the type

```
interface BufferI {
  void put(in T e);
  void get(out T e);
  void increment (unsigned int n);
}
```

is a subtype of type Buffer. BufferI allows clients to send “put” and “get” in arbitrary interleaving. This is possible if messages that BufferI does not want to receive are delayed. All message sequences of Buffer are accepted since the restriction “send get messages only if there are elements in Buffer” is more restrictive than “send get and put messages at any time”. An instance of a subtype always accepts all message sequences of an instance of a supertype.

Previously we examined coordination requirements for a window object. The window can be created and destroyed, iconified and uniconified and partitioned in at most five parts. Although this example may seem simple, in a distributed environment the design of coordination must be made carefully to avoid unintended behavior.

```
interface Window {
  token owner, open, iconified, part;
  Window create() ensure open, part*4;
  Window create_iconified() ensure iconified, part*4;
  void iconify() require open ensure iconified;
  void uniconify() require iconified ensure open;
  Window split_window() require part;
  void close_split_window(in Window w) ensure part;
  void draw_point(in int x,in int y);
  void close() require owner, part*4;
  initial owner
}
```

Four tokens are declared, two of them are used to describe if the window is “open” or “iconified” and one token (“part”) is representing the parts of the window that can be accessed separately. The “owner” token is used to ensure that only one client can close the window. The window object has two initialisers (constructors), the “create” and the “create_iconified” operation. Depending on which initializer was called, the type mark of the window is in the corresponding state, the type of the window is either “Window<owner,open,part∗4>” or “Window<owner,iconified,part∗4>”. A window can be iconified only if it is in the "open" state. If there are two arbitrary objects a and b, it is possible that a creates an instance c of the window object by using the create operation. Object a can pass a reference of c to b, but need not transfer the token "open". Therefore, b is not allowed to iconify the window, (because its type of c is "Window<>"). But it can send "draw_point" messages to c. Object a can send two "split_window" messages to c, which change a's type mark of c to "Window<open,part*2>" and then pass the token "open" and also the two remaining part tokens to b. Therefore a has a reference to c with the type Window. The reference of b to c has the type "Window<open, part*2>". Now neither a nor b can close the window. If b wants to close the window c, it has to wait until a passes the other two "part" tokens to b and it receives the "owner" token. Type combination will combine the tokens of c so that there is a single reference c of type "Window<owner, part*4>". b can close the window because all conditions to close the window are fulfilled.[1]

The following piece of code demonstrates a solution for the dining philosophers problem: A group of philosophers sits around a table and thinks. Sometimes a philosopher becomes hungry and wants to eat spaghetti. For eating this special kind of spaghetti, two forks are needed. But each philosopher has only one single fork, representing a limited shared resource. Before eating, a philosopher has to borrow a second fork from his right neighbour.

```
// IDL-Descripition
interface Fork {
  token up, down;
  void take() require down ensure up;
  void put_down() require up ensure down;
  initial down;
}
interface Phil {
  token asking, nice, noninit;
  void ask(in Phil<asking> prev);
  void yes(in Phil<nice> x, in Fork<down> y) require asking;
  void no() require asking;
  void back(in Fork<down> x) require nice;
  void init(in Phil n) require noninit;
  initial noninit;
}
interface Interruptible {
  void interrupt();
}

// implementation
class CFork : public Fork {
  public:
```

[1] In this case we didn't ensure that there is no object with a reference to c left. It is still possible that an object sends "draw_point" messages to c although c doesn't exist.

```
    void take() { ... }
    void put_down() { ... }
}
class CPhil : public Phil, public Interruptible {
public:
  void init(Phil& n) { next = n; state = thinking; } // store the right neighbor
  void ask(Phil& prev) {
    if (state == thinking) {
      prev.yes(this, myfork); // give the fork to the asking neighbor
      state = nice;
    }
    else prev.no(); // refuse to give the fork
  }
  void yes(Phil& x, Fork& y) {
    nicephil = x; // store the owner of the Fork
    nicefork = y;
    nicefork.take(); //take the forks
    myfork.take();
    state = eating; //start eating
  }
  void no() { state = thinking; }
  void back(Fork& x) { myfork = x; state = thinking; }
  void interrupt() {
    if (state == thinking) { //stop thinking and become hungry
      next.ask(this);
      state = hungry;
    }
    else if (state == eating) { // stop eating and become thinking
      myfork.down(); // drop the forks
      nicefork.down();
      nicephil.back(nicefork); //return the forks
      state = thinking;
    }
  }
private:
  enum {noinit, thinking, nice, hungry, eating} state = noinit;
  Phil& next;
  Phil& nicephil;
  Fork& nicefork;
  CFork myfork;
}
```

A fork can be in two states: Either down on the table, or in the hand of a philosopher. After its creation the fork is down on the table. The interface “Interruptible” is used for sending interrupt messages to a philosopher. We use multiple inheritance to combine the behavior of the "Interruptible" and the "Phil" interface. A philosopher can be in one of three states: "noninit" which assures that the philosopher must be initialised by the message "init", be "nice" by giving the fork to another philosopher and "asking" by asking his neighbor for a fork.

The philosophers are created by an external client. This client has to send "init" to each philosopher and tell about his right neighbor. He can also send interrupts to the philosophers. These interrupts tell a philosopher to change his behavior.

The philosopher has private variables in which he can store his next neighbor. The philosopher who has been nice by giving him his fork, his own fork and the borrowed fork. He also has a state variable keeping information if the philosopher is thinking, nice, hungry or eating. If a philosopher receives an "ask" message, he will borrow the fork by sending "yes" if he is thinking, or he will reply "no". If the philosopher receives "yes", he will take the fork and start eating. If the philosopher receives "no", he will start to think. If a philosopher receives "interrupt" and he is thinking, he will ask his neighbor for the fork and become hungry. If the philosopher stops eating, he will drop both forks and return the borrowed one.

7.3 A Description of the Control Mechanisms for the Process-Type-Checker

The task of the PT-Checker is to check if the conditions on the type mark of an object are fulfilled when a service of an object is invoked. Every object that accesses a server object via a reference has informations about the services the server offers (i.e., argument types, exceptions, etc.) and the possible sequences of calls of these services. These sequences can be controlled by the type mark which is associated with the reference. Since every client can be allowed to access services in a different sequence, the type mark is stored by the client. The messages an object has to accept are controlled by the type mark. The type mark consists of some of the object's tokens; a token can occur more often than once in the type mark.

When an object has been created, the tokens specified in the **initial** statement are contained in the type mark. There is no other way to create or duplicate tokens that are specified in the initial set. Sending a message may require that the sending object (i.e., the client) provides some tokens; they no longer exist after sending the message. These tokens were issued by the object receiving the message (i.e., the server) and passed (in an unspecified way) to the client. When a message is sent, new tokens specified in the corresponding postcondition are issued. Durable tokens can be regarded as an infinite number of equal tokens. The server always knows from the issued tokens which message can be received in the current state. Clients have to coordinate themselves to get the tokens they need.

The set of possible accessible messages of a reference changes according to the type mark. The PT-checker is based on the principle that each reference to an object is associated at run-time with a type mark. The owner of the reference (i.e., the object's client) can use the tokens. The PT-checker has to ensure that the tokens always fulfil the assertions specified by the IDL. After initialisation of an object, the object has no tokens in its type mark. A creator operation must be called which eventually adds some tokens. When the type mark changes, the set of executable operations also changes. The PT-checker has to check that:

- When sending a message to an object through a reference x, the precondition of the corresponding operation must be satisfied. Especially, the type mark of x must contain all required tokens. Furthermore, the assertions and modifiers of arguments and parameters must be compatible.
- When returning results, the type mark of each returned object must be a subtype of the corresponding result type.
- When an operation is executed, the instances' type is updated by removing the non-durable tokens in the precondition and adding the tokens in the postcondition.

Informally spoken, these conditions ensure that all messages satisfy the assertions; this implies that clients send messages and servers change behavior only if they have the required tokens. All type information needed and provided by an object is expressed in the object's parameters and preconditions. For the type checking of the PT-checker, there is no need for it to look beyond interface definitions. If the PT-checker finds an error, an exception is raised, and the message is not sent by the CORBA-ORB. The exception can be handled locally.

7.4 Discussion

Much research on object-oriented type systems was done in theoretical and practical aspects. A large amount of theoretical work was done on object-oriented type systems based on the λ-calculus and the π-calculus [MPW92]. Some work has been done on the idea of increasing the expressiveness of types by adding information about the behavior of the types' instances. The programming language Eiffel [Mey92] and Liskov and Wing's work on behavioral subtyping [LW94] had large influence. Some attempts to introduce methods for code reuse in concurrent systems lead to the idea of decoupling object behavior and object state. Gründer and Geihs propose to add information to the object specification to govern the permissible sequence of method invocations of an object [GG96]. They propose to use Petri-nets, which severely restricts subtyping. In types for concurrent languages it is important to consider the ordering of messages, as in the process type model. An approach quite similar to process types was developed by Najm et al.[NN97]; but this approach supports type splitting only in a very restrictive form, and type combination not at all. Some other concurrent languages like ABCL [Yon90] and concurrent versions of Eiffel [Car93, Mey93] do not handle messages in the order they were sent and received. In these languages, reordering messages is part of the synchronization. Assertions play a very important role in this kind of synchronization. In our proposal, assertions do not delay execution; they just specify required and issued tokens.

There are many approaches to improve the expressiveness of the CORBA-IDL. Bastide et. al. [BSP99] use Cooperative Objects, a dialect of Petri-nets [Pet81] to express behavioral specification. The specification language Larch+ has also been proposed to be integrated in the CORBA-IDL [LC93, Siv95]. Larch+ is based on assertions not powerful enough to express sequences of messages because they are not history-sensitive. Gaspari and Zavattaro model communication between CORBA-objects using a process algebra based on actors [GZ99b]. However, their approach does not deal with types at all, although they model communication protocols.

Not all paradigms of sequential object-oriented programming are useful in distributed object-oriented programming. But the concept of interfaces is quite useful in developing distributed as well as sequential applications. However, especially in distributed systems, the set of messages understood by a server can change during the lifetime of an object. It is necessary that clients coordinate themselves so that they send messages only if and when the server understands the messages. The proposed extension to the CORBA-IDL allows the programmer to express constraints on the ordering of messages accepted by a server. It is possible to introduce a checker in the CORBA-ORB to detect bad messages that may not be understood by the server (or may be understood in an unexpected way). These messages are not sent; instead, exceptions are raised that can be handled locally. The CORBA-interfaces become more expressive, an expressiveness that can be used not only by the compiler, but also by the application designer. Many restrictions on message orderings usually described informally can be described formally in the IDL. Dynamic type checking ensures that these restrictions are actually satisfied. Using the proposed extension of the subtyping relation it is possible for the designer to reuse and extend interfaces in an easy way.

Chapter 8

Conclusion

The concept of strong, static typing is important in software development and is used in many languages like, for example, Eiffel [Mey92], Java [AG96] and C++ [Str97]. In distributed environments, conventional strong, static typing causes difficulties because the set of acceptable messages may change dynamically depending on object states reflecting synchronization conditions. Behavioral types can describe non-uniform behavior: When the state of an object changes, the set of acceptable messages can change, too.

In this theses we dealt with process types [Pun97a], a specific form of behavioral types, and the underlying process calculus [PP99a]. We detected two important requirements on the communication system of the calculus implied by the type concept: Each message can be received by only one object (as in the actors concept), and messages are processed in the same (logical) ordering they were sent. Many important process calculi like the π-calculus [Mil92] and TyCO [Vas99] do not satisfy these requirements. Communication in these calculi occurs via channels; messages may be read from them by several receivers. The ordering of messages is, in general, not preserved.

We gave a fully abstract encoding of our untyped process type calculus in untyped TyCO and vice versa. TyCO is also fully abstract encodable in the π-calculus [Vas94a]. These encodings show that our untyped calculus has the same expressiveness as the π-calculus and TyCO.

Then, we showed that we can express every typed TyCO program in our calculus with process types, but not vice versa. Hence, process types offer more expressiveness than TyCO-types. The process type system supports genericitity and subtyping.

We introduced a notion of interfaces to process types to support self-types. Such self-types are useful to type binary methods. Based on interfaces, we presented a matching relation and a type checking algorithm on interfaces. Our approach to matching resembles that of Abadi and Cardelli for conventional types [AC96a]; we used also subtyping as the basic relation. The results show that the concepts of process types can be adapted to support matching, too.

We looked at CORBA's communication mechanisms and identified possibilities to apply process types to them. We extended the interface description in the IDL by adding tokens as well as pre- and postconditions to specify expected object states (in terms of available tokens). These pre- and postconditions can be checked only dynamically. When the sending of a message violates these conditions, the system raises an exception.

List of Figures

Bibliography

[AC91] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. In *Conference Record of the 18th Symposium on Principles of Programming Languages*, ACM, pages 104–118, 1991. very important, but nearly unreadable.

[AC96a] Martín Abadi and Luca Cardelli. On subtyping and matching. *ACM Transactions on Programming Languages and Systems*, 18(4):401–423, July 1996.

[AC96b] Martín Abadi and Luca Cardelli, editors. *A Theory of Objects*. Springer-Verlag, 1996.

[ACS96] Roberto M. Amadio, Ilaria Castellani, and Davide Sangiorgi. On bisimulations for the asynchronous π-calculus. In *CONCUR '96*, number 1119 in Lecture Notes in Computer Science. Springer-Verlag, 1996.

[AD96] Roberto M. Amadio and Mats Dam. Toward a modal theory of types for the π-calculus. In *Proc. FTRTFT'96,* Lecture Notes in Computer Science, 1135:347–365, 1996.

[AG96] Ken Arnold and James Gosling. *Java*. Addison Wesley, 1996.

[Agh86] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. The MIT Press, 1986.

[AMST92] Gul Agha, Ian A. Mason, Scott Smith, and Carolyn Talcott. Towards a theory of actor computation. In *Proceedings CONCUR'92*, volume 630 of *Lecture Notes in Computer Science*, pages 565–579. Springer-Verlag, 1992.

[BCC+95] Kim Bruce, Luca Cardelli, Giuseppe Castagna, The Hopkins Objects Group, Gary T. Leavens, and Benjamin Pierce. On binary methods. *Theory and Practice of Object Systems*, 1:221–242, 1995.

[BCK93] K. Bruce, J. Crabtree, and G. Kanapathy. An operational semantics for TOOPLE: a statically-typed object-oriented programming language. *Mathematical Foundations of Programming Semantics, Languages and Applications*, 802:215–230, 1993.

[BH91] A. Black and N. Hutchinson. Typechecking polymorphism in Emerald. Technical Report CRL 91/1, DEC Cambridge Research Lab., 1991.

[Boo94] Grady Booch. *Object-Oriented Analysis and Design with Applications.* Benjamin-Cummings, Redwood City, California, second edition, 1994.

[Bou92] Gerard Boudol. Asynchrony and the π-calclus. Technical Report 1702, INRIA, Sophia-Antipolis, 1992.

[Bou97] Gerard Boudol. Typing the use of resources in a concurrent calculus. In *Proceedings ASIAN '97*, number 1345 in Lecture Notes in Computer Science, pages 239–253, Kathmandu, Nepal, 1997. Springer-Verlag.

[BPF97] Kim B. Bruce, Leaf Petersen, and Adrian Fiech. Subtyping is not a good "match" for object-oriented languages. In Mehmet Aksit and Satoshi Matsuoka, editors, *Proceedings ECOOP '97*, number 1241 in Lecture Notes in Computer Science, pages 104–127. Springer-Verlag, 1997.

[Bro97] Hillary Browne. Adding Concurrency to LOOM. Master's thesis, Williams College, 1997.

[Bru94] Kim B. Bruce. A paradigmatic object-oriented programming language: Design, static typing and semantics. *Journal of Functional Programming*, 4(2):127–206, 1994.

[BSP99] Rémi Bastide, Ousmane Sy, and Philippe Palanque. Formal specification and prototyping of CORBA systems. In *ECOOP '99*, Lecture Notes in Computer Science, pages 474–494, Lisboa, Portugal, June 1999. Springer-Verlag.

[BSvG95] Kim B. Bruce, Angela Schuett, and Robert van Gent. Polytoil: A type-safe polymorphic object-oriented language. In *Proceedings ECOOP '95*, number 952 in Lecture Notes in Computer Science, pages 27–51. Springer-Verlag, 1995.

[Car88] Luca Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76:138–164, 1988.

[Car93] Denis Caromel. Toward a method of object-oriented concurrent programming. *Communications of the ACM*, 36(9):90–101, September 1993.

[Cas95] Giuseppe Castagna. Covariance and contravariance: conflict without a cause. *ACM Transactions on Programming Languages and Systems*, 17(3):431–447, 1995.

[CDG+88] L. Cardelli, J. Donahue, L. Galssman, M. Jordan, B. Kalsow, and G. Nelson. Modula-3 report. Technical Report SRC Research Report 31, Digital Equipment Corporation Systems Research Center, 1988.

[Cha92] Craig Chambers. Object-oriented multi-methods in Cecil. In Lehrmann Madsen, editor, *Proceedings ECOOP'92*, volume 615 of *Lecture Notes in Computer Science*, pages 33–56. Springer-Verlag, July 1992.

[CHOM89] P. Canning, W. Hill, W. Olthoff, and J. C. Mitchell. F-bounded polymorphism for object-oriented programming. In *Proceedings of the ACM Conference on Functional Programming and Computer Architecture*, pages 273–280. ACM, 1989.

[CPS97] Jean-Louis Colaço, Marc Pantel, and Patrick Sallé. A set constraint-based analysis of actors. In *Formal Methods for Open Object-based Distributed Systems '97*, July 1997.

[CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, 1985.

[DT88] S. Danforth and C. Tomlinson. Type theories and object-oriented programming. *ACM Computing Surveys*, 20(1):29–72, 1988.

[ES90] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley Publishing Company, Reading, Mass., 1990.

[FG96] Cédric Fournet and Georges Gonthier. The reflexive CHAM and the join-calculus. In *Proceedings ACM-POPL'96*, pages 372–385, 1996.

[Gay93] Simon J. Gay. A sort inference algorithm for the polyadic π-calculus. In *Conference Record of the 20th Symposium on Principles of Programming Languages*, January 1993.

[GG96] H. Gründer and K. Geihs. Reuse and inheritance in distributed object systems. In Bernd Meyer Otto Spaniol, Claudia Linnhoff-Popien, editor, *Trends in Distributed Systems – CORBA and Beyond*, volume 1161 of *Lecture Notes in Computer Science*, pages 191–200. Springer-Verlag, 1996.

[GH98] Andrew D. Gordon and Paul D. Hankin. A concurrent object calculus: Reduction and typing. In *Proceedings HLCL '98*, September 1998.

[GHJV95] A. Gamma, E. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading, 1995.

[GM96] Andreas Gawecki and Florian Matthes. Integrating Subtyping, Matching and Type Quantification: A Practical Perspective. In Pierre Cointe, editor, *Proceedings ECOOP '96*, number 1098 in Lecture Notes in Computer Science, pages 26–47. Springer-Verlag, 1996.

[GZ99a] Mauro Gaspari and Gianluigi Zavattaro. An algebra of actors. In *FMOODS '99*, pages 3–18, Florence, Italy, February 1999. Kluwer Academic Publishers.

[GZ99b] Mauro Gaspari and Gianluigi Zavattaro. A process algebraic specification of the new asynchronous CORBA messaging service. In *ECOOP '99*, Lecture Notes in Computer Science, pages 495–517, Lisboa, Portugal, June 1999. Springer-Verlag.

[HBS73] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal, modular actor formalism for artificial intelligence. *3rd International Joint Conference on Artificial Intelligence*, pages 235–245, 1973.

[Hew77] Carl Hewitt. Viewing control structures as patterns of passing messages. *Journal of Artificial Intelligence*, 8(3), 1977.

[HT91] Kohei Honda and Mario Tokoro. An object calculus for asynchronous communication. In Pierre America, editor, *Proceedings ECOOP'91*, volume 512 of *Lecture Notes in Computer Science*, pages 141–162, Geneva, Switzerland, July 1991. Springer-Verlag.

[KC86] S. N. Khoshafian and G. P. Copeland. Object identity. In *OOPSLA '86*, pages 406–416, September 1986.

[KY94] Naoki Kobayashi and Akinori Yonezawa. Type-theoretic foundations for concurrent object-oriented programming. *ACM SIGPLAN Notices*, 29(10):31–45, October 1994. Proceedings OOPSLA'94.

[Lam78] Leslie Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21:558–564, July 1978.

[LC93] Gary T. Leavens and Yoonsik Cheon. Extending CORBA-IDL to Specify Behavior with Larch. In *OOPSLA '93 Workshop Proceedings: Specification of Behavioral Semantics in OO Information Modeling*, 1993.

[LL99] Charles Lakos and Glenn Lewis. A Practical Approach to Behavioral Inheritance in the Context of Coloured Petri Nets. In *Proceedings SOAP '99*, BRICS Notes Series, Lisboa, Portugal, May 1999. Basic Research in Computer Science.

[LW93] Barbara Liskov and Jeannette M. Wing. Specifications and their use in defining subtypes. *ACM SIGPLAN Notices*, 28(10):16–28, October 1993. Proceedings OOPSLA'93.

[LW94] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.

[Mey92] Bertrand Meyer. *Eiffel: The Language*. Prentice Hall, 1992.

[Mey93] Bertrand Meyer. Systematic concurrent object-oriented programming. *Communications of the ACM*, 36(9):56–80, September 1993.

[Mey94] Bertrand Meyer. *Reusable Software: The Base Object-Oriented Component Libraries*. Prentice-Hall, Englewood Cliffs, NJ, 1994.

[Mil89] Robin Milner. *Communication and Concurrency*. Prentice-Hall, New York, 1989.

[Mil92] Robin Milner. The polyadic π-calculus: A tutorial. In F. L. Bauer, W. Brauer, and H. Schwichtenberg, editors, *Logic and Algebra of Specification*, pages 203–246. Springer-Verlag, 1992.

[MPW92] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes (parts I and II). *Information and Computation*, 100:1–77, 1992.

[MS92] Robin Milner and Davide Sangiorgi. Barbed bisimulation. In *ICALP '92*, number 623 in Lecture Notes in Computer Science. Springer-Verlag, 1992.

[Nel91] Greg Nelson. *Systems Programming with Modula-3.* Series on innovative technology. Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[Nie93] Oscar Nierstrasz. Regular types for active objects. *ACM SIGPLAN Notices*, 28(10):1–15, October 1993. Proceedings OOPSLA'93.

[NN97] Elie Najm and Abdelkrim Nimour. A calculus of object bindings. In *Proceedings of the IFIP Workshop on Formal Methods for Open Object-based Distributed Systems*, Canterbury, United Kingdom, July 1997.

[NN99] Elie Najm and Abdelkrim Nimour. Explicit behavioral typing for object interfaces. In *Proceedings SOAP '99*, BRICS Notes Series, Lisboa, Portugal, May 1999. Basic Research in Computer Science.

[NNS99] Elie Najm, Abdelkrim Nimour, and Jean-Bernad Stefani. Guaranteeing liveness in an object calculus through behavioral typing. In *Proceedings FORTE/PSTV '99*, Bejing, China, October 1999.

[Obj99] Object-Oriented Concepts, Inc. *Orbacus – Manual*, 1999.

[OMG98a] Object Management Group. *The Common Object Request Broker: Architecture and Specification (CORBA)*, 1998. Revision 2.2.

[OMG98b] Object Management Group. *Messaging Service RFP*, 1998.

[OP92] Fredrik Orava and Joachim Parrow. An algebraic verification of a mobile network. *Journal of Formal Aspects of Computing*, 4:497–543, 1992.

[Pet81] James Lyle Peterson. *Petri Net Theory and the Modelling of Systems.* Prentice Hall, 1981.

[Pie94] Benjamin C. Pierce. Bounded quantification is undecidable. *Information and Computation*, 112(1):131–165, July 1994.

[PP97] Christof Peter and Franz Puntigam. Static Type Checking and Deadlock Prevention in Systems Based on Asynchronous Message Passing. In *ECOOP Workshop on Models, Formalisms and Methods for Distributed Object-Oriented Computing*, Finland, June 1997. Electronically available under http://lis.univ-tlse1.fr/ECOOP-WS/Papers/Puntigam/Puntigam.ps.

[PP98] Christof Peter and Franz Puntigam. Coordination of CORBA Objects with Process Types. In *IFIP Middleware '98*, The Lake District, England, September 1998. Electronically available under http://www-sor.inria.fr/-mirrors/middleware98/conference/wips/peter98.ps.gz.

[PP99a] Christof Peter and Franz Puntigam. A concurrent object calculus with types that express sequences. In *Proceedings SOAP '99*, BRICS Notes Series, Lisboa, Portugal, June 1999. Basic Research in Computer Science.

[PP99b] Franz Puntigam and Christof Peter. Static type checking and deadlock prevention in systems based on asynchronous message passing. In *ACM Symposium on Applied Computing*, Lecture Notes in Computer Science, San Antonio, Texas, February 1999. Springer-Verlag.

[PS93] Benjamin C. Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. In *Proceedings LICS'93*, 1993.

[PS97] Benjamin Pierce and Martin Steffen. Higher-order subtyping. *Theoretical Computer Science*, 176(1,2):235–282, 1997. A shorter version appeared in the Proceedings IFIP Working Conference on Programming Concepts, Methods and Calculi (p. 511–530), 1994. Also LFCS technical report ECS-LFCS-94-280 and Interner Bericht IMMD7-01/94, Universität Erlangen.

[PT96] Benjamin C. Pierce and David N. Turner. *Pict: a programming language based on the π-calculus*. University of Cambridge, 1996.

[Pun96a] Franz Puntigam. Synchronization expressed in types of communication channels. In Luc Bougé et al., editor, *Proceedings Euro-Par'96*, volume 1023 of *Lecture Notes in Computer Science*, Lyon, France, August 1996. Springer-Verlag.

[Pun96b] Franz Puntigam. Types for active objects based on trace semantics. In Elie Najm et al., editor, *Proceedings of the 1st IFIP Workshop on Formal Methods for Open Object-based Distributed Systems*, Paris, France, March 1996. IFIP WG 6.1, Chapman & Hall.

[Pun97a] Franz Puntigam. Coordination requirements expressed in types for active objects. In *ECOOP '97*, Lecture Notes in Computer Science, Jyväskylä, Finland, June 1997. Springer-Verlag.

[Pun97b] Franz Puntigam. Types that reflect changes of object usability. In *Proceedings of the Joint Modular Languages Conference*, volume 1204 of *Lecture Notes in Computer Science*, Linz, Austria, March 1997. Springer-Verlag.

[Pun98] Franz Puntigam. Dynamic type information in process types. In David Pritchard and Jeff Reeve, editors, *Proceedings Euro-Par'98*, number 1470 in Lecture Notes in Computer Science, Southampton, England, September 1998. Springer-Verlag.

[Pun99] Franz Puntigam. Non-regular process types. In *EuroPar'99*, Lecture Notes in Computer Science, Toulouse, France, September 1999. Springer-Verlag.

[QW98] Paola Quaglia and David Walker. On encoding pπ in mπ. In *BRICS Report RS–98–26*, BRICS Notes Series, Aarhus University, Denmark, 1998.

[RRV98] António Ravara, Pedro Resende, and Vasco T. Vasconcelos. Towards an algebra of dynamic object types. In *Workshop on Semantics of Objects as Processes*, volume NS–98–5 of *BRICS Notes Series*, June 1998.

[RS95] M. Raynal and M. Singhal. Logical time: A way to capture causality in distributed systems. Technical Report 1472, INRIA, Rennes, 1995.

[RV97] António Ravara and Vasco T. Vasconcelos. Behavioural types for a calculus of concurrent objects. In *Euro-Par'97*, volume 1300 of *Lecture Notes in Computer Science*, pages 554–561. Springer-Verlag, 1997.

[Siv95] Gowri Sankar Sivaprasad. Specifying the Behavior of CORBA-IDL Interfaces. Master's thesis, Department of Computer Science, Iowa State University, December 1995.

[Str97] Bjarne Stroustrup. *The C++ Programming Language.* Addison Wesley Longman, Reading, Mass., 3rd edition, 1997.

[Tes85] Larry Tesler. Object Pascal Report. Technical report, Technical Report 1, 1985.

[Tur92] David N. Turner. *The Polymorphic π-Calculus: Theory and Implementation.* PhD thesis, University of Edinburgh, 1992. ECS-LFCS-96-345.

[Vas94a] Vasco T. Vasconcelos. *A process-calculus approach to typed concurrent objects.* PhD thesis, Keio University, 1994.

[Vas94b] Vasco T. Vasconcelos. Typed concurrent objects. In *Proceedings ECOOP'94*, volume 821 of *Lecture Notes in Computer Science*, pages 100–117. Springer-Verlag, 1994.

[Vas99] Vasco T. Vasconcelos. Processes, functions, and datatypes. *Theory and Practice of Object Systems*, 5(2):97–110, 1999.

[VH93] Vasco T. Vasconcelos and Kohei Honda. Principal typing schemes in a polyadic π-calculus. In *Proceedings CONCUR'93*, July 1993.

[VT93] Vasco T. Vasconcelos and Mario Tokoro. A typing system for a calculus of objects. In *1st International Symposium on Object Technologies for Advanced Software*, volume 742 of *Lecture Notes in Computer Science*, pages 460–474. Springer-Verlag, 1993.

[Weg90] Peter Wegner. Concepts and paradigms of object-oriented programming. *OOPS Messenger*, 1(1):7–87, August 1990.

[WZ88] Peter Wegner and Stanley B. Zdonik. Inheritance as an incremental modification mechanism or what like is and isn't like. In S. Gjessing and K. Nygaard, editors, *Proceedings ECOOP'88*, volume 322 of *Lecture Notes in Computer Science*, pages 55–77. Springer-Verlag, 1988.

[Yon90] Akinori Yonezawa. *ABCL: An Object-Oriented Concurrent System.* The MIT Press, 1990.

[Yos96] Nobuko Yoshida. Graph types for monadic mobile processes. In *16th Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 1180 of *Lecture Notes in Computer Science*, pages 371–386, Hyderabad, India, 1996. Springer-Verlag.

Index

Glossary of Used Symbols

π-Calculus

$P, Q, \ldots$	processes
$a, b, c, \ldots, x, y, z, \ldots$	input names (channels)
$\overline{a}, \overline{b}, \overline{c}, \ldots, \overline{x}, \overline{y}, \overline{z}, \ldots$	output names (co-names)
$\overline{x}\langle y\rangle$	input operation
$x(y)$	output operation
$!P$	replication
$P\|Q$	parallel composition
$P + Q$	summation, selection
$(\nu x)P$	restriction
0	empty process
$P \downarrow_{\overline{a}}, P \Downarrow_{\overline{a}}$	weak commitment, strong commitment
$\overset{\bullet}{\sim}$	strong bisimulation
$\overset{\bullet}{\approx}$	weak bisimulation
$\sim_b$	strong barbed equivalence
$\approx_b$	weak barbed equivalence

Types of Pierce and Sangiorgi

$S, T, U, \ldots$	types
$\Gamma, \Delta, \ldots$	typing assumptions
$(S_1, \ldots, S_n)^I$	channel type
wrong	error

Graph types

$G, G', \ldots$	graphs
$r, s, t \ldots$	base types
Ψ	cut function
$G_1 \odot G_2$	composition
$\alpha, \beta, \ldots$	protocol types

Types of Quaglia and Walker

$\mathcal{S}, \ldots$	types
λ	typing
$\sigma, \tau, \ldots$	m-types
$\mathcal{G}_\lambda$	labelled directed graph

concς-Calculus

$p, q, r \ldots$	names
$x, y, z, \ldots$	variables

$l, \ldots$	labels
$p.l$	method selection
$[l_i = \varsigma(x_i)b_i^{i\in 1..n}]$	denomination
$u.l \Leftarrow \varsigma(x)b$	method update
$a \upharpoonright b$	(not commutative) parallel composition
$(\nu p)a$	restriction
$A, B, \ldots$	type
$A <: B$	subtype relation

TyCO

$a, b, c, \ldots, u, \ldots$	names
X, Y, Z	abstraction variables
$l, \ldots$	labels
$P, Q, \ldots$	processes
$A, B, \ldots$	process abstractions
$M, \ldots$	method collections
$a!l{:}[\tilde{v}]$	message
$a?M$	object
$P \vert Q$	parallel composition
$(\nu x)P$	scope restriction
$\mathbf{def}\ X_1 = A_1\ \mathbf{in}\ P$	declaration
$\{l_1{:}A_1\& \ldots \& l_n{:}A_n\}$	method collection
$X[\tilde{v}]$	applied abstraction variable
$A[\tilde{v}]$	applied abstraction
$a{*}?\{l_1{:}(\tilde{x}_1)P_1\& \ldots\}$	replication

Types of Vasconcelos

$t, t', \ldots$	type variables
$\alpha, \beta, \ldots$	polymorphic types
$x : \alpha$	type assignment
Γ_T, Δ_T	typings
$\sigma, \ldots$	polymorphic types

Behavioral types of Ravara and Vasconcelos

$[]$	hole
$\mathcal{T}$	set of types
$\alpha^b, \beta^b, \gamma^b, \ldots$	types
V_{α^b}	set of nodes of type α^b
I_{α^b}	set of initial nodes of type α^b
S_{α^b}	set of arcs of type α^b
Γ	typing
$\asymp$	compatibility

$\uplus$	union
$\Vert$	interleaving

Najm and Nimour's OL-Calculus

$u[m_1(\tilde{I}_1) = B_1, \ldots]$	object
$u.m(\tilde{\rho}\tilde{v}) > B$	send operation
$u + v$	message selection
$E \triangleright rx$	type
$\sim_N$	bisimulation

Process Type Calculus

$x, y, z, \ldots$	names
$l, \ldots$	labels
$P, Q, \ldots$	objects
$\mu, \nu, \ldots$	messages
$\pi, \rho, \sigma, \tau, \ldots$	process types
$\alpha, \beta, \gamma, \ldots$	message descriptors
$a, b, c, \ldots$	tokens
$\varphi, \ldots$	message prototype
$p, q, \ldots$	sequences of prototypes
$\kappa, \ldots$	interfaces
$\chi, \ldots$	exact interface
$\psi, \ldots$	not exact interface
$a\langle B/\tilde{\mu}\rangle$	object with message queue
$l(\tilde{v})$	message
$l(\underline{\tilde{x}})B$	creator method
$\boldsymbol{\{}\tilde{M}\boldsymbol{\}}$	selection of methods
$a \triangleleft \mu.B$	send operation
$(\underline{a})@\mu.B$	object creation
$@\mu$	become operation
$\{\tilde{M}\}$	program
$\eta_{\{\tilde{M}\}}$	system configuration
$\eta_{\{\tilde{M}\}} \xrightarrow{a} \theta_{\{\tilde{M}\}}$	execution relation
$\boldsymbol{\{}\tilde{\alpha}\boldsymbol{\}}$	static part of a type
$[\tilde{a}]$	dynamic part of a type
$\boldsymbol{\{}\tilde{\alpha}\boldsymbol{\}}[\tilde{a}]$	process type
$(\underline{t})\boldsymbol{\{}\tilde{\alpha}\boldsymbol{\}}[\tilde{a}]$	recursive process type
$\tau_1 \vert \tau_2$	parallel composition
$x(\underline{\tilde{s}})(\tilde{\tau})[\tilde{a}_1][\tilde{a}_2]$	message descriptor
$seq(\tau)$	sequencing function
$\sharp\boldsymbol{\{}\tilde{\alpha}\boldsymbol{\}}[\tilde{a}]$	exact interface

$x(\underline{\tilde{s}})(\tilde{\kappa})[\tilde{a}_1][\tilde{a}_2]$	message interface
$T(\kappa, \{\tilde{\alpha}\})$	substitution function
$\asymp$	configuration equivalence
$\equiv$	structural type equivalence
$\cong$	type equivalence
$\gg$	redundancy relation
$\leq$	subtyping relation
$\doteqdot$	structural interface equivalence
$\backsimeq$	interface equivalence
$\leqslant$	interface matching relation

CURRICULUM VITAE

Date of Birth: July 7, 1973

Place of Birth: Hohenems

Education:

1979-1983 Primary school
1983-1991 Secondary school BRG Dornbirn Schoren
June 1991 Graduated from Secondary school with Matura (A-levels)
1991-1996 Studies of Computer Science at the Technical University of Vienna
June 1996 Graduated as a Dipl.-Ing. (Master of Science) at the Technical University of Vienna
Thesis on "Typechecking of active objects with Process Types"
since 1996 PhD Student in Computer Science

Professional Experience:

1997-1998 Software Engineer for Simulations
1999 Research Assistant at the Institut für Computersprachen, Vienna
1999-2000 National Service at the diocese Feldkirch, Vorarlberg

Publications:

Christof Peter, "Typüberprüfung von aktiven Objekten mit Prozeßtypen", master's thesis, Technical University Vienna, 1996, in german.

Christof Peter, Franz Puntigam, "Static Type Checking and Deadlock Prevention in Systems Based on Asynchronous Message Passing", ECOOP Workshop on Models, Formalisms and Methods for Distributed Object-Oriented Computing, June 1997, Finland.

Christof Peter, Franz Puntigam, "Coordination of CORBA Objects with Process Types", IFIP Middleware '98, September 1998, The Lake District, England.

Christof Peter, Franz Puntigam, "Static Type Checking and Deadlock Prevention in Systems Based on Asynchronous Message Passing", ACM Symposium on Applied Computing, February 1999, LNCS, San Antonio, Texas, Springer Verlag.

Christof Peter, "Application of Process Types", Workshop report PhDOOS 99, June 1999, Lisboa, Portugal.
Also in PhDOOS '99 Workshop proceedings, June 1999, Lisboa, Portugal.

Christof Peter, Franz Puntigam, "A Concurrent Object Calculus with Types that Express Sequences", Proceedings SOAP '99, BRICS Notes Series, June 1999, Lisboa, Portugal.