On the Mechanized Validation of Infinite-State and Parameterized Reactive and Mobile Systems

Christine Röckl

Lehrstuhl für Informatik VII
der Technischen Universität München

On the Mechanized Validation of Infinite-State and Parameterized Reactive and Mobile Systems

Christine Röckl

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:	Univ.-Prof. Tobias Nipkow Ph.D.
Prüfer der Dissertation:	1. Univ.-Prof. Dr. Javier Esparza
	2. Univ.-Prof. Dr. Dr. h.c. Wilfried Brauer
	3. Prof. Davide Sangiorgi, INRIA Sophia-Antipolis/Frankreich

Die Dissertation wurde am 06.11.2000 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 02.02.2001 angenommen.

Abstract

The growing influence of telecommunication-systems in all areas has brought with it the need for elaborate and reliable software running on concurrent and, in particular, dynamically changing systems. With the development of such systems becoming ever more complex, the outline and development of mechanized and mechanizable verification-techniques is indispensable. This overall goal can be divided into three tasks: (1) outline of a framework in which (2) complex systems can be analysed, and the (3) design and implementation of efficient proof-techniques. It is certainly beyond the scope of a thesis to provide an integrated framework dealing with these questions within one environment. This work therefore concentrates on dominant aspects of each of the three tasks in separate discussions. A major result of the thesis is that the approaches outlined for each of the three questions obviously fit together, so that they can be used as a basis for the development of an integrated framework. Often an automatic verification of larger hardware- and software-applications is impossible, because the systems are too large or even infinite. This thesis focusses on infinite-state and parameterized systems, and builds tool-support on interactive theorem-proving. Further, it concentrates on the validation of correct implementations by exploiting behavioural reasoning. It demonstrates how to apply various techniques to tackle proofs about infinite-state and parameterized systems with large descriptions, discusses the support that theorem-provers can offer, and presents a foundational platform for reasoning about mobile systems in a general-purpose theorem-prover.

Several design-decisions have to be taken. This thesis uses process-algebra as a framework, in particular CCS for the description and analysis of infinite-state reactive systems, and the π-calculus for the discussion of mobile and higher-order systems. For a validation, it applies observation-equivalence, exploiting a range of proof-techniques that have been developed for the two calculi. The contributions of the thesis with respect to the three verification-tasks are as follows.

(1) A formalization of the π-calculus in Isabelle/HOL is presented. It uses a *shallow embedding*, in which α-conversions and β-reductions on names bound by input and restriction are dealt with by a λ-calculus provided by Isabelle. A shallow embedding has the advantage that substitutions do not have to be defined and applied in a semantic analysis of the processes, like in a *deep embedding*; implementing substitutions in theorem-provers usually is a tedious task and prone to errors. On the other hand, syntax-analysis in shallow embeddings is intricate, because structural induction fails; further, exotic terms arise from an application of operators that do not belong to the π-calculus, in definitions of the continuations of an input or restriction. This thesis discusses how to mimic structural induction by rule-induction over a *well-formedness* predicate on processes which simultaneously rules out exotic terms. It discusses proof-techniques based on instantiations and re-abstractions of functions over processes,

and uses them to derive vital syntactic properties of the π-calculus. Finally, it proves that the shallow embedding is fully adequate with respect to a straightforward deep embedding.

(2) The use of the π-calculus to give semantics to higher-order imperative concurrent languages is studied. The thesis presents a translation of Concurrent Idealized Algol (CIA) into the π-calculus, which integrates previous encodings of imperative and concurrent features into CCS, and such of functional elements into the π-calculus. It is proved by exhibiting an operational correspondence that the encoding is sound with respect to a straightforward small-step operational semantics of CIA employing a bisimulation-based operational congruence. The argument makes extensive use of proof-techniques such as bisimulation up to expansion and contextual reasoning. The encoding is then applied to validate classical benchmark laws and examples for CIA via a translation into the π-calculus. Further, the thesis presents a correctness-proof for a more involved example employing procedures of higher order. This example emphasizes that the translation of programs from a higher-order syntax (CIA) into a first-order syntax (π-calculus) allows for a validation of examples that cannot be treated otherwise. Yet, even in the π-calculus an application of involved proof-techniques—in particular, bisimulations up to expansion and context—is indispensable.

(3) A report on the mechanization in Isabelle/HOL of bisimulation-proofs for infinite-state and parameterized systems is given. The proofs follow a direct approach by exhibiting relations and proving that they are bisimulations, or bisimulations up to expansion. This methodology is obviously well-suited for a mechanization, because it consists of two clearly separated parts: the conceptually difficult part is to set up the relation, whereas the proof that it is indeed a bisimulation falls into a potentially large number of simple and often uniform steps. The first part has to be performed by the user, while the second part can be carried out in interaction with the tool. The prover automatically deals with all aspects of bookkeeping, and is usually able to deal with simple cases fully automatically. Complex cases require an intervention by the user; yet, as usually most cases are similar, this task can be alleviated by designing suitable proof-procedures. The thesis discusses the mechanization of bisimulation-proofs on case-studies from hardware-verification using CCS as a semantic model: an infinite-state version of the Alternating-Bit-Protocol is validated, as well as a specification of the Sliding-Window-Protocol; further, proofs of Write-Invalidate Memory-Cache-Coherence are discussed, one employing a standard bisimulation, and the other applying a bisimulation up to expansion.

Contents

Contents **vii**

1 Introduction **1**

2 Preliminaries **5**
2.1 CCS 6
2.1.1 Syntax and Semantics 6
2.1.2 Value-Passing CCS 8
2.1.3 Observation-Equivalence 10
2.2 The π-Calculus 14
2.2.1 Syntax and Semantics 15
2.2.2 Bisimilarity 18
2.2.3 Type-Systems 19
2.3 General-Purpose Theorem-Proving 20
2.3.1 Isabelle/HOL 21
2.3.2 Representing Languages with Binders 29

3 Formalizing the π-Calculus in HOAS **35**
3.1 Formal Syntax 36
3.1.1 A First-Order Syntax 37
3.1.2 A Higher-Order Syntax 46
3.2 Deriving Syntactic Properties 50
3.2.1 Monotonicity 51
3.2.2 Extensionality of Contexts 52
3.2.3 β-Expansion 53
3.3 Adequacy 57
3.3.1 The Encoding-Function 58
3.3.2 The Decoding-Function 59
3.3.3 Proving Adequacy 59
3.4 Discussion 60

4 Encoding Algol in the π-calculus **63**
4.1 Concurrent Idealized Algol 64
4.1.1 CIA-{**await**} 66
4.1.2 Full CIA 67
4.1.3 Observational Congruence 70
4.2 Encoding Concurrent Idealized Algol 71

4.2.1 Encoding CIA-{**await**} . . . 72
4.2.2 Encoding Full CIA . . . 78
4.3 Laws and Examples . . . 83
4.3.1 Verification of a Higher-Order Buffer . . . 85
4.4 Discussion . . . 88

5 Mechanized Validation of Infinite-State Systems 91
5.1 Observation-Equivalence wrt. L . . . 93
5.1.1 Faulty Channels . . . 94
5.1.2 The Alternating Bit Protocol . . . 95
5.2 Compositionality . . . 98
5.2.1 A Sliding-Window Protocol . . . 99
5.3 Up-To Techniques . . . 102
5.3.1 Write-Invalidate Cache-Coherence . . . 103
5.4 Discussion . . . 108

6 Conclusion and Future Work 111

Bibliography 115

Index 121

Acknowledgements

During the last four years, many people have influenced my life and work. First of all, I would like to thank my supervisors, Javier Esparza and Davide Sangiorgi, for accompanying this thesis. Many of the ideas discussed in this work are due to them. Their insistence on clarity and precision have made a great impression on me, and have certainly influenced the development and presentation of this thesis. They have given me enduring support, and have taken my occasional stubbornness with patience. Further, I would like to express my gratitude to my head of chair, Wilfried Brauer, for creating an inspiring atmosphere in which I was able to dare my first steps into research and teaching without having to bother about material questions. I have always marvelled at his devotion and knowledge, and feel grateful for having him as a teacher and mentor.

Many thanks to the following people for inspiring discussions and suggestions: Paul Attie, Joëlle Despeyroux, Daniel Hirschkoff, Anna Ingolfsdottir, Kedar Namjoshi, Tobias Nipkow, and Carlos Puchol. I have especially enjoyed the joint exploration of the mysteries of logic with Daniel. Thanks to my colleagues, Markus Holzer, Astrid Kiehn, Barbara König, Tony Kucera, Angelika Mader, Jürgen Menden, Stephan Melzer, Leonor Prensa-Nieto, Stefan Römer, Peter Rossmanith, Claus Schröter, Stefan Schwoon, and Frank Wallner, for a lot of fun and fruitful discussions.

I would like to thank my family for their prevailing support and comprehension, even if some of my decisions must have been hard to swallow for them. It is a good feeling to know that they stand by my side without compromise. Many thanks also to my friends for giving me a brilliant time. I am particularly happy to know Christine Duus, Monika Federle, Sandra Hischier, Sybille Rummler, and Axel Ziganki.

Chapter 1

Introduction

The growing influence of telecommunication-systems has brought with it the need for elaborate, flexible, and reliable software running on parallel, distributed, and dynamically changing systems. With the design of these systems becoming ever more complex, highly involved description-techniques have emanated: programming-languages combine imperative, object-oriented, and functional features with concurrency. Programs written in them are usually large and produce an infinite number of states. Tool-supported verification of these sophisticated systems is indispensable, yet automatic techniques are often not applicable due to state-explosion or undecidability problems. It is therefore necessary to design interactive techniques that are general enough for the analysis of infinite-state and parameterized reactive and mobile systems, but are practically applicable offering an intuitive style of reasoning and a maximal degree of automation. The outline and implementation of a complete framework is certainly beyond the scope of a thesis, but should be considered as a milestone for future research. The work at hand represents a first step towards this direction by addressing three dominant questions behind this objective. A main result of this thesis is that the techniques we have chosen and discuss in more or less separate investigations, obviously fit together in a natural way; we are therefore confident that the material discussed and developed in this work can serve as the basis of an integrated framework.

Issues in validation The three questions are as follows. (1) A language for the description of systems has to be provided by a verification-platform. We choose the interactive theorem-prover Isabelle/HOL [Pau94, Pau93] as the foundation of a mechanization, and implement the π-calculus [MPW92, Mil99] in it. Our choice of Isabelle/HOL is motivated by the fact that it offers a human-style way of reasoning, which is essential in interactive proofs. In particular, Isabelle/HOL allows for a definition and application of concrete syntax which considerably enhances readability, and offers a range of proof-strategies including induction, case-analysis, as well as tactics dealing with 'obvious' cases fully automatically. Further, it contains large databases with predefined data-structures and theorems derived for them. For instance, we make extensive use of the theories concerning sets and lists. Our choice of the π-calculus is motivated by its simplicity and expressive power: it has a first-order syntax, because in communications references are exchanged instead of entire processes; nevertheless it can describe higher-order systems, modelling functions by using references. This simplicity certainly comes for a price: structural information, such as local distribution, is completely lost. On the other hand, it is exactly this simplicity which makes effective proofs possible, and allows us to derive

a syntactic framework within Isabelle/HOL. (2) In order to derive proofs about systems, they have to be translated into the description-language provided by the verification-framework. Writing programs directly in description-languages like the π-calculus is tedious; for larger applications, where structural information is essential to maintain the code, it is completely illusive. On the other hand, verifying programs in higher-order languages is hardly possible. In fact, for higher-order languages with local variables it is even unclear how to define practically applicable criteria for telling that two program-phrases are equivalent. For this purpose, one usually considers the phrases within all possible contexts. This certainly yields a means of telling apart phrases by exhibiting a distinguishing context; to obtain an effective validation-methodology, on the other hand, one has to be able to dispose of the universal quantification over the contexts. We explore a compromise, proposing to write programs in a high-level language and to translate them into a description-language afterwards in order to allow for their validation. As a description-language, we choose the π-calculus again. We investigate its applicability by giving a π-calculus semantics to Concurrent Idealized Algol (CIA) [Rey81, Bro96]. Our choice of CIA is due to the fact that it combines imperative, functional, and concurrent features in an elegant way, and has been the object of extensive study. Because CIA has a higher-order syntax—it contains a call-by-name λ-calculus—proofs of program-equivalences are hard, and are further aggravated by the concept of local and global variables. A translation into the π-caluclus allows us to derive proofs of program-equivalences that would otherwise not be possible. Our approach is in line with work on giving game-semantics to sequential Idealized Algol [AM96a, AM99], translating program-phrases into a syntactically simple operational model in order to exploit the proof-techniques developed for it. (3) In our proofs that two programs or systems are equivalent, we follow an operational approach, employing observation-equivalence [Par80, Mil89]. In order to show that an implementation matches its specification, we exhibit a relation and prove that it is a bisimulation. This approach is popular in process-algebra, and has been particularly applied and analysed in the context of CCS and the π-calculus. Bisimulation-proofs are universally applicable to systems with finite and infinite state-spaces alike, and are well-suited for a mechanization in interactive theorem-provers. A bisimulation-proof can be divided into two parts: the first part is to exhibit a relation containing as a pair the system and its specification, and is conceptually difficult; the second part is to check for each pair (P, Q) in the relation that Q can simulate every step $P \xrightarrow{\alpha} P'$ of P by a sequence of steps $Q \xrightarrow{\tau} \dots \xrightarrow{\tau} \xrightarrow{\alpha} \xrightarrow{\tau} \dots \xrightarrow{\tau} Q'$ (where the τ-steps denote internal activities) such that (P', Q') is again in the relation, and vice versa. The second part usually amounts to a tedious case-study, and often many of the cases are similar. In a mechanized proof of observation-equivalence, the user sets up a relation and then derives in interaction with the theorem-prover that it is a bisimulation. The theorem-prover completely keeps track of the bookkeeping, telling the user the bisimulation-obligations that still have to be proved. Further, it can usually deal with the simple cases fully automatically, and the user can specify proof-procedures for the more complex cases in order to exploit similarities in the derivations of matching sequences of steps. As a consequence, the bisimulation-proof is not only machine-checked but can be easier to deal with than a related proof on paper. We mechanize a number of case-studies in Isabelle/HOL using CCS in order to describe both implementations and specifications of infinite-state and parameterized reactive systems; similar proofs for mobile systems in the π-calculus can be formalized analogously.

Infinity This thesis focusses on the validation of infinite-state systems. There are several potential sources of infinity. (1) Programs employ data of an infinite type. In some cases, one can abstract from these data and obtain finite descriptions that can be treated by fully automatic techniques, but especially higher-order programs often involve complex data-structures that are essential for their semantics, and therefore have to be taken into account. (2) Often programs have the ability to fork processes. There is no general means of telling a priori how many of these tasks will be created during a run, and usually programs are even assumed to fork arbitrarily many of them. Such programs have to be considered as structurally infinite. In this case, it is impossible to obtain a finite abstraction. (3) Further, a system can consist of an unspecified but finite number of components. Such a parameterization is often considered even harder to treat than data-infinity or structural infinity, or simply an infinite number of (uniform) components executing in parallel. The reason is that one has to keep track of a finite yet unspecified number of components.

We consider infinite-state systems of all three kinds, investigating into bisimulation-techniques for them. Data-infinity turns out to be the easiest case. Also, parameterization can be treated surprisingly well: the parameter becomes part of the relation, and a bismulation-proof considers an unspecified relation $\mathcal{R}_n$ from the family $\bigcup_n \mathcal{R}_n$ ranging over all parameters n. The hardest case with respect to proofs of observation-equivalence is structural infinity. Reasoning about systems with the capability of forking processes usually involves proof-techniques like bisimulations up to expansion and contextual reasoning. The former to abstract from internal administrative activities such as the rearrangement of components, or garbage-collection; the latter can be used to deal with the forked components if they are similar for both processes to be compared. Structural infinity is a major issue in higher-order languages like CIA.

Outline of the Thesis

This thesis is organized as follows. In Chapter 2, we present preliminary material, on which this work is based. In particular, we give an introduction to CCS, the π-calculus, and interactive theorem-proving in Isabelle/HOL, with emphasis on the formalization of CCS and the π-calculus. In Chapter 3, we discuss in greater detail the issue of embedding the π-calculus in Isabelle/HOL. We provide an embedding which frees the user from an explicit treatment of binders by using higher-order abstract syntax. In Chapter 4, we present a translation of CIA into the π-calculus, and use it to validate benchmark laws and examples as well as a more complex examples involving procedures of higher order. In Chapter 5, we present a number of case-studies on the mechanization of proofs of observation-equivalence in Isabelle/HOL using CCS. Chapter 6 concludes with an evaluation of the material presented in this thesis and gives directions for future work.

Chapter 3 A formalization of the π-calculus in Isabelle/HOL is presented. It uses a *shallow embedding*, in which α-conversions and β-reductions on names bound by input and restriction are dealt with by a λ-calculus provided by Isabelle. A shallow embedding has the advantage that substitutions do not have to be defined and applied in a semantic analysis of the processes, like in a *deep embedding*; implementing substitutions in theorem-provers usually is a tedious task and prone to errors. On the other hand, syntax-analysis in shallow embeddings is intricate, because structural induction fails; further, exotic terms arise from an application of operators that do not belong to the π-calculus, in definitions of the continuations of an input

or restriction. This thesis discusses how to mimic structural induction by rule-induction over a *well-formedness* predicate on processes which simultaneously rules out exotic terms. It discusses proof-techniques based on instantiations and re-abstractions of functions over processes, and uses them to derive vital syntactic properties of the π-calculus. Finally, it proves that the shallow embedding is fully adequate with respect to a straightforward deep embedding.

Chapter 4 The use of the π-calculus to give semantics to higher-order imperative concurrent languages is studied. The thesis presents a translation of Concurrent Idealized Algol (CIA) into the π-calculus, which integrates previous encodings of imperative and concurrent features into CCS, and such of functional elements into the π-calculus. It is proved by exhibiting an operational correspondence that the encoding is sound with respect to a straightforward small-step operational semantics of CIA employing a bisimulation-based operational congruence. The argument makes extensive use of proof-techniques such as bisimulation up to expansion and contextual reasoning. The encoding is then applied to validate classical benchmark laws and examples for CIA via a translation into the π-calculus. Further, the thesis presents a correctness-proof for a more involved example employing procedures of higher order. This example emphasizes that the translation of programs from a higher-order syntax (CIA) into a first-order syntax (π-calculus) allows for a validation of examples that cannot be treated otherwise. Yet, even in the π-calculus an application of involved proof-techniques—in particular, bisimulations up to expansion and context—is indispensable.

Chapter 5 A report on the mechanization in Isabelle/HOL of bisimulation-proofs for infinite-state and parameterized systems is given. The proofs follow a direct approach by exhibiting relations and proving that they are bisimulations, or bisimulations up to expansion. This methodology is obviously well-suited for a mechanization, because it consists of two clearly separated parts: the conceptually difficult part is to set up the relation, whereas the proof that it is indeed a bisimulation falls into a potentially large number of simple and often uniform steps. The first part has to be performed by the user, while the second part can be carried out in interaction with the tool. The prover automatically deals with all aspects of bookkeeping, and is usually able to deal with simple cases fully automatically. Complex cases require an intervention by the user; yet, as usually most cases are similar, this task can be alleviated by designing suitable proof-procedures. The thesis discusses the mechanization of bisimulation-proofs on case-studies from hardware-verification using CCS as a semantic model: an infinite-state version of the Alternating-Bit-Protocol is validated, as well as a specification of the Sliding-Window-Protocol; further, proofs of Write-Invalidate Memory-Cache-Coherence are discussed, one employing a standard bisimulation, and the other applying a bisimulation up to expansion.

Accompanying Material

Much of the material presented in this thesis is formalized in Isabelle/HOL. The proof-scripts are available at `http://www7.in.tum.de/~roeckl/thesis/`. Some results of this thesis are discussed in [RHB00, RS99, Röc99, RE99, Röc00, RE00].

Chapter 2

Preliminaries

Classical process algebras like CCS, ACP, or CSP [Mil89, BW90, Hoa85] model *static* systems, where the topology of the systems does not change during execution. The classical notions of CCS and ACP abstract from data, and have been the basis for a large field of foundational studies. These have led to a broad theory, covering both the *operational*—transition systems, behavioural equivalences and preorders—and the algebraical—algebraic rules, syntactic transformations of process-terms—approaches. Extensions of CCS and ACP, as well as the original notion of CSP, are equipped with the possibility to handle data, and have been successfully applied in the modelling and verification of reactive systems in concurrent frameworks. In particular, they are a standard model in hardware verification.

With the introduction of mobile telecommunication, the need arose for a model of *dynamic*, or *mobile*, systems, where the topology, that is the binding structure, changes during execution. For this purpose, Milner, Parrow, and Walker, proposed the π-calculus [MPW92, Mil99], which extends the process agebra CCS with a concept of *references* to processes that can be passed around so to make previously unknown components accessible. This concept gives the π-calculus the potential to model software systems as well. It has been proved in early works already to be a faithful model of higher-order languages like the λ-calculus [Chu40, Bar81], see [Mil92b], or object-oriented languages [Wal95].

CCS and the π-calculus can be fruitfully applied in the validation of hardware and software. A popular approach is to model both implementation and specification of a system in terms of processes and to apply *observation-equivalence* [Par80, Mil89] to show that they behave in accordance with one another. For larger systems, tool-support is indispensable; in the case of infinite-state systems we propose the use of interactive theorem-provers. In this chapter, we introduce background material for the thesis. Section 2.1 introduces CCS, and describes in terms of an example how to show in an operational argument that two systems are observation-equivalent. In Section 2.2, we present the π-calculus and describe how bisimulations for it can be made coarser by applying type-systems. The use of type-systems in the π-calculus is common: bisimulations are often too discriminating, because the calculus abstracts from structural information; type-systems help to restore it. Section 2.3 describes general-purpose theorem-proving, demonstrating the application of Isabelle/HOL in the context of process-algebra. In particular, we discuss the use of higher-order abstract syntax (HOAS) in the formalization of languages with binders.

P	$::=$	0	Inaction
	$\mid$	$\alpha.P$	Prefix, where $\alpha \in \mathcal{A}ct$
	$\mid$	$P\backslash L$	Restriction, over a set of labels $L \subseteq \mathcal{L}$
	$\mid$	$P[f]$	Relabelling, with $f : \mathcal{L} \to \mathcal{L}$ such that $f(\bar{a}) = \overline{f(a)}$
	$\mid$	$P + P$	Nondeterministic Choice
	$\mid$	$P \mid P$	Parallel Composition
	$\mid$	A	Agent, specified in an agent definition $A \stackrel{\text{def}}{=} P$

Table 2.1: **Syntax of CCS.** The set $\mathcal{P}_{\text{CCS}}$ of CCS-processes is built on a constant for inaction, 0, by applying prefixing, restriction, choice, and parallel composition. Infinite behaviour can be obtained from agent definitions in which the denominator A occurs on the right-hand side of the defining equation $A \stackrel{\text{def}}{=} P$.

2.1 CCS

The *Calculus of Communicating Systems (CCS)* was introduced by Milner as a foundational model of nondeterministic and concurrent systems [Mil89]. In its *pure* form, CCS completely abstracts from any form of data, hence the usual *synchronous point-to-point communication* between two processes reduces to their *synchronization*. In *value-passing* CCS, processes exchange data in a communication; nevertheless, the theory can be adapted from that of pure CCS.

2.1.1 Syntax and Semantics

The Calculus of Communicating Systems centres on the synchronization of *processes* running in parallel. Consider two processes $\bar{a}.P$ and $a.Q$; the first can perform an output on a channel a, continuing with the behaviour of P afterwards, whereas the second can perform a corresponding input on a, continuing like Q. This is formally expressed in terms of *labelled transitions* $\bar{a}.P \xrightarrow{\bar{a}} P$ and $a.Q \xrightarrow{a} Q$. When run in parallel, the two processes thus have the possibility to communicate on channel a, producing the *invisible action* τ in the transition $\bar{a}.P \mid a.Q \xrightarrow{\tau} P \mid Q$.

Names, labels, actions Let $\mathcal{N}$ be a countably infinite set of *channels*, or *names*, ranged over by $a, b, \ldots$. An *output* on a name a is denoted by marking it with a bar, yielding $\bar{a}$. For an *input*, the names are generally used in their pure form. Often, $\bar{a}$ is referred to as the *complement* or *co-name* of a, and the resulting set of co-names $\overline{\mathcal{N}}$ is ranged over by $\bar{a}, \bar{b}, \ldots$. It is assumed that $\mathcal{N} \cap \overline{\mathcal{N}} = \emptyset$. Further, for the sake of symmetry, complementation is usually assumed to be idempotent; that is, $\bar{\bar{a}} = a$. The set of visible *labels* unifies names and co-names, $\mathcal{L} \stackrel{\text{def}}{=} \mathcal{N} \cup \overline{\mathcal{N}}$, and is ranged over by $l, m, \ldots$. Together with the *invisible (or, silent) action* τ, the labels yield the set of *actions*, $\mathcal{A}ct \stackrel{\text{def}}{=} \mathcal{L} \cup \{\tau\}$. We use $\alpha, \beta, \ldots$ to range over $\mathcal{A}ct$.

Constants and combinators The set $\mathcal{P}_{\text{CCS}}$ of CCS-processes is built on a constant 0 for *inaction* that cannot exhibit any kind of behaviour, by the following operators: a *prefix* process $\alpha.P$ behaves like P after an execution of the action α; a *restriction* $P\backslash L$ can be used to force

$$\frac{}{\alpha.P \stackrel{\alpha}{\longrightarrow} P}\ \mathbf{C1} \qquad \frac{P \stackrel{\alpha}{\longrightarrow} P' \qquad \alpha, \bar{\alpha} \notin L}{P\backslash L \stackrel{\alpha}{\longrightarrow} P'\backslash L}\ \mathbf{C2} \qquad \frac{P \stackrel{\alpha}{\longrightarrow} P'}{P + Q \stackrel{\alpha}{\longrightarrow} P'}\ \mathbf{C3a} \qquad \frac{P \stackrel{\alpha}{\longrightarrow} P'}{P \stackrel{f[\alpha]}{\longrightarrow} P'}\ \mathbf{C4}$$

$$\frac{P \stackrel{\alpha}{\longrightarrow} P'}{P \mid Q \stackrel{\alpha}{\longrightarrow} P' \mid Q}\ \mathbf{C5a} \qquad \frac{P \stackrel{l}{\longrightarrow} P' \qquad Q \stackrel{\bar{l}}{\longrightarrow} Q'}{P \mid Q \stackrel{\tau}{\longrightarrow} P' \mid Q'}\ \mathbf{C6} \qquad \frac{P \stackrel{\alpha}{\longrightarrow} P' \qquad A \stackrel{\text{def}}{=} P}{A \stackrel{\alpha}{\longrightarrow} P'}\ \mathbf{C7}$$

Table 2.2: **Operational Semantics of CCS.** The *labelled transition-system* for $\mathcal{P}_{\text{CCS}}$ is the least set of triples $P \stackrel{\alpha}{\longrightarrow} P'$ described by the rules **C1**–**C7**. Rules **C3b** and **C5b** are symmetric versions of **C3a** and **C5a**, so we omit them here. In rule **C7**, we use $l \in \mathcal{L}$ to range over all kinds of labels, that is, for both names and co-names In Rule **C4**, we extend f to $\mathcal{A}ct$ by postulating $f(\tau) = \tau$.

that certain channels can only be used in internal synchronizations; a *relabelling* $P[f]$ renames channels by means of a function f with $f(\bar{a}) = \overline{f(a)}$; the *choice* process $P + Q$ can choose between behaving like P or like Q; the *composed* processes $P \mid Q$ run in parallel; finally, *agents* $A \stackrel{\text{def}}{=} P$ allow for the modularization of a process-descriptions and for the introduction of infinite behaviour. Table 2.1 shows a formal description of the syntax of CCS.

Operational semantics We follow an *operational* approach, describing the behaviour of $\mathcal{P}_{\text{CCS}}$ in terms of *labelled transition-rules*, rather than giving rules for process-transformation, as it is usual in the *algebraic*, or *syntactical*, approach. A *transition* $P \stackrel{\alpha}{\longrightarrow} P'$ is a triple, where P and P' are processes describing the states before and after the transition, and the action α Table 2.2 defines a *labelled transition-system* for CCS, in terms of an *inductive* set of rules; that is, in terms of a least fixpoint. This means that the rules completely describe the behaviour of $\mathcal{P}_{\text{CCS}}$; that is, if a transition $P \stackrel{\alpha}{\longrightarrow} P'$ cannot be derived for a process P, the process cannot perform it. As a consequence, the *introduction-rules* **C1**–**C7** in Table 2.2 yield a corresponding *elimination-rule* performing *case-injection*, that is, telling for a transition how it must have been derived.

Note that *interleaving* semantics like that of CCS interpret parallelism as a form of non-determinism: given a transition of a parallel composition $P \mid Q \stackrel{\alpha}{\longrightarrow} R$, it either stems from a transition of P or Q, or from a communication between them, with $R = P' \mid Q'$ for suitable derivatives P' and Q'.

Consider again the processes $\bar{a}.P$ and $a.Q$. Their transitions $\bar{a}.P \stackrel{\bar{a}}{\longrightarrow} P$ and $a.Q \stackrel{a}{\longrightarrow} Q$ are derivable directly from the axiom **C1**. Further, a simple—however tedious—case-injection yields that no other transition is possible, because none of the transition-rules would apply. The synchronization $\bar{a}.P \mid a.Q \stackrel{\tau}{\longrightarrow} P \mid Q$ can finally be derived by an application of rule **C5** for communication.

Weak transitions *Strong transitions* $P \stackrel{\alpha}{\longrightarrow} P'$ give equal relevance to each step of a system; the operational semantics as given by rules **C1**–**C7** in Table 2.2 is *small-step*, so to capture nondeterminism in the behaviour of the processes. *Weak transitions* $P \stackrel{\alpha}{\Longrightarrow} P'$ abstract from internal communications and silent prefixes, because often one is interested in the *observable* behaviour rather than considering every step a system makes. For example, consider a sequence of transitions $P \stackrel{\tau}{\longrightarrow} P_1 \stackrel{\tau}{\longrightarrow} P_2 \stackrel{a}{\longrightarrow} P_3 \stackrel{\tau}{\longrightarrow} P_4 \stackrel{\tau}{\longrightarrow} P'$; if we are not interested in the number

of silent steps occurring before and after the a-transition, we can reduce it to $P \overset{a}{\Longrightarrow} P'$.

Formally, let $P \overset{\epsilon}{\Longrightarrow} P'$ be defined by Kleene's star $P(\overset{\tau}{\longrightarrow})^* P'$. Further, for every $\alpha \in \mathcal{A}ct$, the transition $P \overset{\alpha}{\Longrightarrow} P'$ abbreviates the chain $P \overset{\epsilon}{\Longrightarrow} P_1 \overset{\alpha}{\longrightarrow} P_2 \overset{\epsilon}{\Longrightarrow} P'$. As a notational convention, we further write $P \overset{\hat{\alpha}}{\Longrightarrow} P'$ to denote $P \overset{\alpha}{\Longrightarrow} P$ for visible $\alpha \in \Lambda$, and $P \overset{\epsilon}{\Longrightarrow} P'$ for $\alpha = \tau$. From this definition, we can derive weak transition laws, such as the following:

$$\frac{}{P \overset{\epsilon}{\Longrightarrow} P}\ \textbf{Cw1} \qquad \frac{P \overset{\alpha}{\longrightarrow} P'}{P \overset{\alpha}{\Longrightarrow} P'}\ \textbf{Cw2a} \qquad \frac{P \overset{\alpha}{\longrightarrow} P'}{P \overset{\hat{\alpha}}{\Longrightarrow} P'}\ \textbf{Cw2b} \qquad \frac{P \overset{\epsilon}{\Longrightarrow} P_1 \overset{\alpha}{\Longrightarrow} P_2 \overset{\epsilon}{\Longrightarrow} P'}{P \overset{\alpha}{\Longrightarrow} P'}\ \textbf{Cw3}$$

Proving them is a standard exercise in induction on the length of the weak transitions in the premises of the laws.

Further, there exist weak versions of all rules in Table 2.2. Versions in which α is replaced by $\hat{\alpha}$ exist as well, except for rules **C3a** and **C3b** concerning choice. The reason is that they are not monotonic; for instance, in order to derive $P + Q \overset{\alpha}{\Longrightarrow} P'$ from $P \overset{\alpha}{\Longrightarrow} P'$, at least one action does have to take place, in order to eliminate Q, and this cannot be guaranteed by a weak ϵ-transition.

2.1.2 Value-Passing CCS

In *value-passing CCS (VP)*, labels in Λ are considered to contain values of some type such as integers or booleans, or parameters for which values can be introduced. An output-label $\bar{a}(\tilde{v})$ is interpreted to send a list of values $\tilde{v} = v_1, \ldots, v_n$ along a channel a. An input-label $a(\tilde{x})$, on the other hand, is able to receive a list of values $\tilde{v}$ along channel a; the list $\tilde{x}$ determines formal parameters for which $\tilde{v}$ can be introduced. We call a the *subject*, and $\tilde{v}$ respectively $\tilde{x}$ the *objects* of the labels.

Early and late transitions As a concrete example, consider the processes $\bar{a}(3, tt, 7).P$ and $a(x, y, z).Q$. In a communication, the first process transmits the values 3, tt, and 7, to the second, yielding a transition,

$$\bar{a}(3, tt, 7).P \,|\, a(x, y, z).Q \overset{\tau}{\longrightarrow} P \,|\, Q\{3/x, tt/y, 7/z\}.$$

Adopting an *early* view on input, according to which input-labels are immediately instantiated in a transition, we can say that it is based on the two transitions,

$$\bar{a}(3, tt, 7).P \overset{\bar{a}(3,\, tt,\, 7)}{\longrightarrow} P \qquad \text{and} \qquad a(x, y, z).Q \overset{a(3,\, tt,\, 7)}{\longrightarrow} Q\{3/x, tt/y, 7/z\}.$$

This contrasts *late* semantics, where instantiations are delayed until an actual communication happens. There, the communication from above would be derived applying substitution only together with the rule for communication, from the two transitions,

$$\bar{a}(3, tt, 7).P \overset{\bar{a}(3,\, tt,\, 7)}{\longrightarrow} P \qquad \text{and} \qquad a(x, y, z).Q \overset{a(x,\, y,\, z)}{\longrightarrow} Q.$$

Remark: We consider an early semantics for value-passing CCS more natural than a late one. The reason is that in VP there is a clear distinction between values and variables used as formal parameters in the continuation Q of an input prefix $a(\tilde{x}).Q$. To illustrate this with an example, a transition,

$$a(\tilde{x}).Q \overset{a(\tilde{x})}{\longrightarrow} Q, \qquad \text{(in contrast to an early transition } a(\tilde{x}).Q \overset{a(\tilde{v})}{\longrightarrow} Q\{\tilde{v}/\tilde{x}\})$$

always yields an *open* term as a derivative; that is, a term containing variables $\tilde{x}$ that are not bound by an input-prefix. We shall see later that in the π-calculus, a late semantics is as natural as an early one, because there is no nominal distinction between names and variables; see Section 2.2.1 for more details.

Operational semantics A labelled transition-system in early style for VP can be obtained from that for pure CCS in Table 2.2 by slightly modifying rules **C1** and **C2** as follows:

$$\frac{}{\tau.P \xrightarrow{\tau} P}\ \textbf{C1a} \qquad \frac{}{\bar{a}(\tilde{v}).P \xrightarrow{\bar{a}(\tilde{v})} P}\ \textbf{C1b} \qquad \frac{}{a(\tilde{x}).P \xrightarrow{a(\tilde{v})} P\{\tilde{v}/\tilde{x}\}}\ \textbf{C1c}$$

$$\frac{P \xrightarrow{\tau} P'}{P\backslash N \xrightarrow{\tau} P'\backslash N}\ \textbf{C2a} \qquad \frac{P \xrightarrow{\bar{a}(\tilde{v})} P' \qquad a \notin N}{P\backslash N \xrightarrow{\bar{a}(\tilde{v})} P'\backslash N}\ \textbf{C2b} \qquad \frac{P \xrightarrow{a(\tilde{v})} P' \qquad a \notin N}{P\backslash N \xrightarrow{a(\tilde{v})} P'\backslash N}\ \textbf{C2c}$$

Remark: As there is a clear distinction between input and output in VP, we can assume in Rules **C2b** and **C2c** that restriction is performed over sets of channels.

As for Rule **C6** concerning communication, we consider $a(\tilde{v})$ and $\bar{a}(\tilde{v})$ to complement one another. In an early semantics, the rule can thus be used without modification.

Mapping VP to CCS Reducing VP to pure CCS is usually not easy, because the values often belong to infinite types, or the length of the object-vectors are not limited. If they are, however, each instantiation $a(\tilde{v})$ of an input-prefix $a(\tilde{x})$ can be considered as a name, and each corresponding output-prefix $\bar{a}(\tilde{v})$ can be regarded as the respective co-name. With $\mathcal{V}$ the type of the values, an input-prefix $a(\tilde{x}).Q$ can then be written as a nondeterministic choice over all possible instantiations; that is, $\sum_{\tilde{v}\in\tilde{\mathcal{V}}} Q\{\tilde{v}/\tilde{x}\}$. Similarly, the sets of labels belonging to a restriction have to be modified by adding adequate objects to the channels they contain. In this case, the theory of pure CCS can be directly transferred to VP. Note that VP over an infinite type of values yields a labelled transition-system with infinite branching, which makes a finite axiomatization impossible.

Modelling Systems In practice, one often does not write down systems in terms of process-terms, but provides transition-rules describing their behaviour together with rules **C2**–**C7** from Table 2.2. As an example, consider the systems $(P_1(0)\,|\,C())\backslash\{c\}$ and $(P_2(0)\,|\,C())\backslash\{c\}$, consisting of producers P_1 and P_2, and a consumer C. The producers send consecutive natural numbers to the consumer C on a channel c, which is made private to the two parallel components by a restriction. We assume P_1 to work correctly, but P_2 and C to be faulty: from time to time, they spontaneously loose values in a silent transition. The behaviour of the components can be described operationally as follows:

$$P_1(i) \xrightarrow{\bar{c}(i)} P_1(i+1),\quad i \geq 0 \qquad\qquad \begin{array}{l} P_2(i) \xrightarrow{\bar{c}(i)} P_2(i+1),\quad i \geq 0 \\ P_2(i) \xrightarrow{\tau} P_2(i+1),\quad i \geq 0 \end{array} \qquad\qquad \begin{array}{l} C(s) \xrightarrow{c(x)} C(sx) \\ C(xs) \xrightarrow{\bar{a}(x)} C(s) \\ C(s_1 x s_2) \xrightarrow{\tau} C(s_1 s_2) \end{array}$$

Remark: The two producers can easily be modelled in terms of (infinite families of) recursive agents $P_1(i) \stackrel{\text{def}}{=} \bar{c}(i).P_1(i+1)$ and $P_2(i) \stackrel{\text{def}}{=} \bar{c}(i).P_2(i+1) + \tau.P_2(i+1)$ as well. Giving a monolithic definition of the consumer is a bit clumsier, because its behaviour is determined by the structure of the list of values it manipulates.

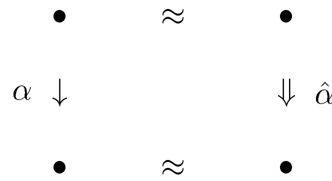

Table 2.3: **Observation-Equivalence:** *Weak bisimilarity*, or *observation equivalence*, contains pairs of systems that are capable of *simulating* the behaviour of one another. Its definition consists of a *transitional*—an action performed by one process has to be simulated by the other—and a *closure*—the derivatives have to belong to the relation—part. A formal description is given in Definition 2.1.

Infinity The two systems are *infinite-state*, that is, they yield transition graphs with an infinite number of nodes. Infinity can have various sources. (1) Process P_1, for example, employs values of an infinite type, and so does P_2; we therefore call them *data-infinite*. The consumer need not necessarily do so; it can as well be attached to a producer $P_3 \stackrel{\text{def}}{=} \bar{c}(0).P_3$, for instance. (2) Yet, the consumer has the potential to store as many values as it likes; it is therefore *structurally infinite*. Another example—from pure CCS—of structural infinity is the agent $A \stackrel{\text{def}}{=} a.0 \,|\, \tau.A$, adding a component $a.0$ with each silent transition. (3) A third source of infinity is *parameterization*, where a system can consist of one, five, or even seven-hundred, copies of a component; the actual number is then usually replaced by an abstract parameter n. As an example, consider $(P_1(0) \,|\, \ldots \,|\, P_1(0) \,|\, C())\backslash\{c\}$, where the number n of producers clearly determines the behaviour of the system: of each number $i \in \mathbb{N}$, at most n copies will be delivered to the environment; that is, as many as there are copies of P_1 in the system.

2.1.3 Observation-Equivalence

Interleaving semantics of concurrent systems are characterized by a *branching* behaviour of the systems, which should be taken into account by suitable notions of equivalence. A standard example for the weaknesses of language equivalence in CCS are the processes $a.(b.0 + c.0)$ and $a.b.0 + a.c.0$, both producing the language $\{\epsilon, a, ab, ac\}$. Nevertheless, the first process only decides after the a-transition whether to perform a b or a c, whereas the second one has to decide immediately. *Bisimulation-equivalences* have been introduced by Park and Milner in order to capture exactly the branching aspects in the behaviour of concurrent systems [Par80, Mil89].

Weak bisimilarity, or *observation-equivalence*, is the bisimulation-equivalence that is most suitable for the comparison of concrete systems modelled in CCS, because it abstracts over silent transitions, nevertheless without discarding the branching branching aspects in the behaviour of the systems. Like other notions of bisimilarity, it is defined as a relation $\approx$ over pairs of processes which can simulate single actions of one another (*transitional* part) yielding derivatives which are again in the relation (*closure* part). This is illustrated by the diagram in Table 2.3. A bisimulation proof thus reduces to making the diagram commute. Recall from Section 2.1.1 above that $\hat{\alpha}$ denotes α for visible $\alpha \in \mathcal{L}$, and ϵ for $\alpha = \tau$.

Definition 2.1 (Observation Equivalence) A relation $\mathcal{R} \subseteq \mathcal{P}_{\text{CCS}} \times \mathcal{P}_{\text{CCS}}$ is a *weak bisimulation*, if for all $P\mathcal{R}Q$ and all $\alpha \in \mathcal{A}ct$, the following holds:

(i) if $P \stackrel{\alpha}{\longrightarrow} P'$, then there exists some Q' such that $Q \stackrel{\hat{\alpha}}{\Longrightarrow} Q'$ and $P'\mathcal{R}Q'$;

(ii) if $Q \xrightarrow{\alpha} Q'$, then there exists some P' such that $P \stackrel{\hat{\alpha}}{\Longrightarrow} P'$ and $P'\mathcal{R}Q'$.

Observation equivalence is defined as the union of all weak bisimulations; that is, $\approx \stackrel{\text{def}}{=} \bigcup\{\, \mathcal{R} \mid \mathcal{R} \text{ is a weak bisimulation} \,\}$. Hence, $P \approx Q$ if there exists a weak bisimulation containing (P, Q).

Compositionality An interesting property of observation-equivalence is that it is *a congruence* with respect to parallel composition, and restriction; that is, $P \approx Q$ implies $P \mid R \approx Q \mid R$ and $P\backslash L \approx Q\backslash L$, for arbitrary $R \in \mathcal{P}_{\text{CCS}}$ and $L \subseteq \mathcal{L}$. This means that a verification of systems based on observation-equivalence is *compositional*: properties of systems can easily be derived from suitable properties of their specifications. We discuss this topic in more detail in Chapter 5.

In pure CCS, it is further a congruence with respect to prefix; that is, $P \approx Q$ implies $\alpha.P \approx \alpha.Q$, for all $\alpha \in \mathcal{A}ct$. In value-passing CCS, compositionality can be fully assumed for silent and output-prefixes; for input-prefixes over open terms, it further has to be assumed that two open terms are bisimilar if they are bisimilar with respect to all instantiations.

Remark: With the clear distinction between open and closed terms in VP (see also Section 2.1.2), this universal quantification naturally yields compositionality also for input-prefixes. By contrast, the fact that this distinction is less clear in the π-calculus, necessitates an additional reasoning about substitutions.

Comparing systems Bisimulation-equivalences immediately yield a *semantically oriented* methodology for proving that two systems are bisimilar: it suffices to exhibit a suitable relation containing the two systems as a pair and prove that it is a bisimulation. An alternative approach is *syntactically oriented*: applying *algebraic* rules, one transforms one system into the other. Often, the algebraic approach is considered to be more elegant, because it reasons fully on a syntactic level. On the other hand, it severely suffers from the *state-explosion problem*: during a transformation, parallel compositions are unfolded into summations (that is, choice), which yields a term of exponential size in the number of parallel compositions. The problem is that it is a single term that explodes. By contrast, in a semantical proof, it is the number of proof obligations that explodes; as each of them can easily be treated separately, the (modularized) proof can be kept manageable.

As an example, consider again the systems $(P_1(0) \mid C())\backslash\{c\}$ and $(P_2(0) \mid C())\backslash\{c\}$ from the previous section. We would like to establish that lossiness on part of the producer does not matter to the environment if the consumer looses values as well. In order to show that the systems are observation-equivalent, we prove that the relation,

$$\mathcal{R}_{pc} \stackrel{\text{def}}{=} \{\, ((P_1(i) \mid C(s))\backslash\{c\},\ (P_2(i) \mid C(s))\backslash\{c\}) \ \mid\ i \in I\!N \wedge s \in I\!N^n \wedge n \geq 0 \,\}$$

is a weak bisimulation. Clearly, $\mathcal{R}_{pc}$ contains the two systems as a pair. Considering a pair $(P_1(i) \mid C(s))\backslash\{c\}\, \mathcal{R}_{pc}\, (P_2(i) \mid C(s))\backslash\{c\}$, we have to examine all possible transitions of the two components, with respect to the transition as well as closure part:

- P_1 produces a value, in a transition $(P_1(i) \mid C(s))\backslash\{c\} \xrightarrow{\tau} (P_1(i{+}1) \mid C(si))\backslash\{c\}$. Clearly, P_2 can do the same, yielding $(P_2(i) \mid C(s))\backslash\{c\} \xrightarrow{\tau} (P_2(i+1) \mid C(si))\backslash\{c\}$. Obviously, the resulting derivatives again yield a pair in $\mathcal{R}_{pc}$.

- P_2 produces a value, in a transition $(P_2(i) \,|\, C(s))\backslash\{c\} \xrightarrow{\tau} (P_2(i+1) \,|\, C(si))\backslash\{c\}$. This case is fully symmetrical.

- P_2 looses a value, yielding $(P_2(i) \,|\, C(s))\backslash\{c\} \xrightarrow{\tau} (P_2(i+1) \,|\, C(s))\backslash\{c\}$. In this case, P_1 can make use of the lossiness of C, by sending a value to the consumer so that it can be disposed of there; hence, $(P_1(i) \,|\, C(s))\backslash\{c\} \xrightarrow{\tau} (P_1(i+1) \,|\, C(si))\backslash\{c\} \xrightarrow{\tau} (P_1(i+1) \,|\, C(s))\backslash\{c\}$. Obviously, the resulting derivatives again yield a pair in $\mathcal{R}_{pc}$. In this case, we have exploited the possibility of choosing the weak transition of the answering process, by constructing a convenient sequence of transitions.

- C delivers a value, yielding $(P_1(i) \,|\, C(ks))\backslash\{c\} \xrightarrow{\bar{a}(k)} (P_1(i) \,|\, C(s))\backslash\{c\}$ on the one hand, and $(P_2(i) \,|\, C(ks))\backslash\{c\} \xrightarrow{\bar{a}(k)} (P_2(i) \,|\, C(s))\backslash\{c\}$ on the other. Clearly, the resulting derivatives again yield a pair in $\mathcal{R}_{pc}$.

- C looses a value, yielding $(P_1(i) \,|\, C(s_1ks_2))\backslash\{c\} \xrightarrow{\tau} (P_1(i) \,|\, C(s_1s_s))\backslash\{c\}$ on the one hand, and $(P_2(i) \,|\, C(s_1ks_2))\backslash\{c\} \xrightarrow{\tau} (P_2(i) \,|\, C(s_1s_s))\backslash\{c\}$ on the other. Clearly, the resulting derivatives again yield a pair in $\mathcal{R}_{pc}$.

Refinements The attentive reader will have noticed that $\mathcal{R}_{pc}$ contains more pairs than are actually reachable for the systems, because it does not further specify the list s stored in C. In order merely to prove $(P_1(0) \,|\, C())\backslash\{c\} \approx (P_2(0) \,|\, C())\backslash\{c\}$, this imprecision does not matter, yet even simplifies the bisimulation-proof. A more precise bisimulation can be obtained by *refining* the relation $\mathcal{R}_{pc}$, that is, by adding more conditions the pairs in the relation have to fulfill. The resulting relation $\mathcal{R}'_{pc}$ then is a proper subset of the original relation:

$$\mathcal{R}'_{pc} \stackrel{\text{def}}{=} \{ ((P_1(i) \,|\, C(s))\backslash\{c\},\, (P_2(i) \,|\, C(s))\backslash\{c\}) \;|\; \begin{array}{l} i \in I\!N \wedge s \in I\!N^n \wedge n \geq 0 \wedge \\ s(n) < i \wedge \\ \forall j.\, 0 \leq j < n \Rightarrow s(j) < s(j+1) \,\} \end{array}$$

Like $\mathcal{R}_{pc}$ above, $\mathcal{R}'_{pc}$ is a weak bisimulation. The proof is similar, yet in each obligation, the additional monotonicity-properties for s have to be considered. As a result, monotonicity of the objects of the visible transitions of the systems can be inferred directly from the bisimulation-relation.

Showing inequivalence Bisimulation-equivalences are determined by an alternation of a universal—*for all* $P\mathcal{R}Q$ and all $\alpha \in \mathcal{A}ct$—and an existential—*there exists* some Q'/P'—parts. As a result, a bisimulation proof can be considered as a *game* between an *attacker* who wants to prove that two given systems are *not* bisimilar suitably choosing from the universal part, and a *defender* trying to maintain that the systems *are* bisimilar selecting adequate answers from the existential part. If two systems are not bisimilar, there exists a finite winning strategy for the attacker; see [Ste97] for a discussion.

The method of exhibiting a bisimulation-relation is rather suited for validating that two systems are equivalent. The construction of a bisimulation is usually an incremental process: one starts with a kernel of the relation and adds pairs of processes until all states of the systems and all their transitions are considered by the relation. In the case where two systems are not bisimilar, one occasionally ends up with a pair that does not produce matching transitions. From such a pair one usually can reconstruct the cause of the failure, and correct the systems

accordingly or prove their inequivalence by designing, from the setup of the relation, a proof-strategy for the above attacker.

Expansion Like above, we use $\hat{\alpha}$ to denote α for visible actions $\alpha \in \Lambda$, and ϵ for $\alpha = \tau$. Recall that $P \stackrel{\epsilon}{\Longrightarrow} P'$ means that P performs arbitrarily many—including zero—silent transitions in order to become P'. In analogy, we define $P \stackrel{\epsilon}{\longrightarrow} P'$ as a transition of zero or one step; that is, either $P' = P$ or $P \stackrel{\tau}{\longrightarrow} P'$.

We are now ready to refine observation-equivalence by considering efficiency in the behaviour of the processes. Intuitively, a process P *expands* a process Q, written $P \geq Q$, if $P \approx Q$ but P needs less silent steps to execute than Q [AKH92].

Definition 2.2 (Expansion) A relation $\mathcal{R} \subseteq \mathcal{P}_{\mathrm{CCS}} \times \mathcal{P}_{\mathrm{CCS}}$ is an *expansion-relation*, if for all $P\mathcal{R}Q$ and all $\alpha \in \mathcal{A}ct$, the following holds:

(i) if $P \stackrel{\alpha}{\longrightarrow} P'$, then there exists some Q' such that $Q \stackrel{\hat{\alpha}}{\longrightarrow} Q'$ and $P'\mathcal{R}Q'$;

(ii) if $Q \stackrel{\alpha}{\longrightarrow} Q'$, then there exists some P' such that $P \stackrel{\alpha}{\Longrightarrow} P'$ and $P'\mathcal{R}Q'$.

Expansion is defined as the union of all expansion relations; that is, $\geq \stackrel{\mathrm{def}}{=} \bigcup\{\mathcal{R} \mid \mathcal{R}$ is an expansion relation$\}$. Hence, $P \geq Q$ if there exists an expansion-relation containing (P, Q).

Consider again $(P_1(0) \mid C())\backslash\{c\}$ and $(P_2(0) \mid C())\backslash\{c\}$ from Section 2.1.2. In the proof that the two systems are bisimilar, we have made use of the fact that the first system can compensate for P_2 loosing values by sending values to C and dispose of them there. All other transitions are mapped one-by-one. This suggests that $\mathcal{R}_{pc}$ is not only a weak bisimulation, but an expansion establishing $(P_1(0) \mid C())\backslash\{c\} \geq (P_2(0) \mid C())\backslash\{c\}$.

Proof-techniques Recall from above that observation equivalence is *compositional*. This can be exploited in bisimulation-proofs of large compound systems. Consider an abstract system $(P \mid Q)\backslash L$ with a specification S. Assume further that the process P is very large, but that it is observation-equivalent to a much smaller process P'. The congruence-results for parallel composition and restriction allow us to conclude that $(P \mid Q)\backslash L \approx S$ provided $(P' \mid Q)\backslash L \approx S$. Proving the second condition is clearly much easier, because with P' being much smaller than P, less proof-obligations have to be considered.

Besides compositionality, there exists a technique to reduce the size of the relation—and hence the number of proof-obligations to examine—by considering a subset $\mathcal{R}$ of a weak bisimulation only. In that case, closure—that is, the condition $P'\mathcal{R}Q'$ from Definition 2.1—is proved *up to* a suitable notion of equivalence or preorder. An adequate preorder supporting observation equivalence is expansion, as defined above.

Definition 2.3 (Bisimulation up to Expansion) A relation $\mathcal{R} \subseteq \mathcal{P}_{\mathrm{CCS}} \times \mathcal{P}_{\mathrm{CCS}}$ is a *weak bisimulation up to expansion*, if for all $P\mathcal{R}Q$ and all $\alpha \in \mathcal{A}ct$, the following holds:

(i) if $P \stackrel{\alpha}{\longrightarrow} P'$, then there exist some P'', Q', Q'' such that $Q \stackrel{\hat{\alpha}}{\Longrightarrow} Q'$ and $P' \geq P''\mathcal{R}Q'' \leq Q'$;

(ii) if $Q \stackrel{\alpha}{\longrightarrow} Q'$, then there exist some P' such that $P \stackrel{\hat{\alpha}}{\Longrightarrow} P'$ and $P' \geq P''\mathcal{R}Q'' \leq Q'$.

It is a standard exercise to show that two processes P and Q are observation equivalent if there exists a bisimulation up to expansion containing (P, Q).

The *up to* proof technique was introduced by Milner in [Mil89], and has been further investigated by Milner and Sangiorgi in [SM92, San95]. Expansion is a *progressing* relation, that is, processes become 'smaller' by its application [San95]. The use of expansion instead of observation-equivalence in up-to proofs is necessitated by the fact that observation-equivalence itself does not progress, that is, it does not necessarily force processes to act. This is precarious when applying it in up-to proofs, as pointed out in [San95]. As an example, consider the relation

$$\mathcal{R}_{\not\approx} \stackrel{\text{def}}{=} \{(\tau.a.0,\ 0)\}$$

which is a bisimulation up to observation equivalence, although obviously $\tau.a.0 \not\approx 0$. This counterexample was established independently by Sangiorgi, Sjödin, and Jonsson.

2.2 The π-Calculus

The π-calculus [MPW92, Mil99] is a value-passing language based on CCS, and is particularly characterized by the fact that in its basic form subject and objects of an input or output are all of the same type, $\mathcal{N}$ for names. This implies that a name received in a communication can be used as a subject later on; like channel x in the following example:

$$!(ay.\bar{y}c.0) \mid \bar{a}x.xz.P \;\xrightarrow{\tau}\; !(ay.\bar{y}c.0) \mid \bar{x}c.0 \mid xz.P \;\xrightarrow{\tau}\; !(ay.\bar{y}c.0) \mid 0 \mid P\{c/z\}.$$

Process $!(ay.\bar{y}c.0)$ can be considered as a simple procedure, which upon each call along its 'address' a with a 'return channel' y, delivers a copy of a constant name c on y. The other process, $\bar{a}x.xz.P$, asks the procedure to return its result along a channel x, possibly to make use of it later on. In order to ensure that the result is indeed returned to the caller—and not to some interfering process running in parallel—the π-calculus allows the client to send a private, that is, restricted, name to the procedure:

$$!(ay.\bar{y}c.0) \mid (\nu x)(\bar{a}x.xz.P) \;\xrightarrow{\tau}\; (\nu x)(!(ay.\bar{y}c.0) \mid \bar{x}c.0 \mid xz.P);$$

which yields by α-equivalence,

$$\equiv_\alpha \; !(ay.\bar{y}c.0) \mid (\nu x)(\bar{x}c.0 \mid xz.P) \;\xrightarrow{\tau}\; !(ay.\bar{y}c.0) \mid (\nu x)(0 \mid P\{c/z\}).$$

In this example, the scope of the local variable x is extended from the client to an instance of the procedure during execution. This *mobility*—that is, the fact that the binding structure of a process can change during its execution—is due to the syntactic simplicity of the π-calculus, by not making any distinction between channels and values sent along them.

Descriptive Power It is exactly this simplicity which yields the descriptive power of the π-calculus. As indicated by the above example, the π-calculus can model procedure calls, even with private parameters, such as local variables. In fact, as we will discuss in Chapter 4, the π-calculus is a faithful model of concurrent imperative programming languages of higher order, such as Concurrent Algol. Further, as pointed out soon after the introduction of the π-calculus [MPW92] already, mobility suffices to describe higher-order processes [Tho90, San92, Ama93, San96a] and functions [Mil92b, San96a]. The work in Chapter 4 especially builds on the results for higher-order functions.

P	$::=$	0	Inaction
	$\mid$	$\alpha.P$	Prefix, where $\alpha \in \mathcal{A}ct$
	$\mid$	$(\nu\tilde{x})P$	Restriction, over a finite list of names $\tilde{x} \subset \mathcal{N}$
	$\mid$	$P + P$	Nondeterministic Choice
	$\mid$	$P \mid P$	Parallel Composition
	$\mid$	$[a = b]P$	Matching, with names $a, b \in \mathcal{N}$
	$\mid$	$[a \neq b]P$	Mismatching, with names $a, b \in \mathcal{N}$
	$\mid$	$!P$	Replication

Table 2.4: **Syntax of the π-Calculus.** The set $\mathcal{P}$ of π-calculus processes is built on a constant for inaction, 0, by applying prefixing, restriction, choice, parallel composition, matching, mismatching, and replication.

2.2.1 Syntax and Semantics

In this section, we consider a *polyadic* π-calculus, in which objects are tuples of names. The *monadic* calculus, in which every object consists of a single name, can be considered as a special case. In fact, polyadicity does not add any expressive power; that is, the polyadic π-calculus can be expressed in terms of the monadic one [Yos96, QW98].

Names, labels, actions As for CCS, we consider a countably infinite set $\mathcal{N}$ of *names*, ranged over by $a, b, x, y, \ldots$. For a better distinction of names that are *free* in a process and those that are *bound*, we use $a, b, \ldots$ for the former, and $x, y, \ldots$ for the latter. The set $\mathcal{L}$ of visible *labels* contains *inputs* $a(\tilde{x})$ and *free outputs* $\bar{a}\tilde{b}$. We refer to a as the *subject* of the labels, and to $\tilde{x}$ and $\tilde{b}$ as their objects. *Actions* in $\mathcal{A}ct$ are given by labels and the *invisible (or, silent) action* τ, and are ranged over by $\alpha, \beta, \ldots$. Besides, there is a *bound output* $(\nu\tilde{x})\bar{a}\tilde{b}$ used on transition arrows, telling that those names in $\tilde{b}$ that also occur in $\tilde{x}$, are private.

In the *monadic* π-calculus, every object tuple is of length one; that is, the labels are of the form $\bar{a}b$ for free output, and ax for input (sometimes, also $a(x)$). A bound output $(\nu x)\bar{a}x$ is often written as $\bar{a}(x)$, in analogy to input.

Constants and combinators The set $\mathcal{P}$ of π-calculus processes is built on a constant 0 for inaction that cannot exhibit any kind of behaviour, by the following operators: a *prefix* process $\alpha.P$ behaves like P after an execution of the action α; a *restriction* $(\nu\tilde{x})P$ can be used to make certain names *local* or *private*; the *choice* process $P + Q$ can choose between behaving like P or like Q; the *composed* processes $P \mid Q$ run in parallel; *matching* $[a = b]P$ and *mismatching* $[a \neq b]P$ implement a conditional on the equality of names; finally, the *replicated* process $!P$ behaves like an arbitrary number of copies of P put in parallel. Table 2.4 shows a formal description of the syntax of the π-calculus.

The π-calculus is particularly characterized by its two binders input and restriction. Especially restriction is responsible for the structure of a process, by creating objects which can communicate on private channels, and can make previously private channels available to other processes as well.

Free and bound names *Free* and *bound* names are a predominant issue in the π-calculus.

$$\begin{array}{rcl@{\qquad}rcl}
fn(0) & \stackrel{\text{def}}{=} & \emptyset & bn(0) & \stackrel{\text{def}}{=} & \emptyset \\
fn(\tau.P) & \stackrel{\text{def}}{=} & fn(P) & bn(\tau.P) & \stackrel{\text{def}}{=} & bn(P) \\
fn(\bar{a}\tilde{b}.P) & \stackrel{\text{def}}{=} & \{a,\tilde{b}\} \cup fn(P) & bn(\bar{a}\tilde{b}.P) & \stackrel{\text{def}}{=} & bn(P) \\
fn(a\tilde{x}.P) & \stackrel{\text{def}}{=} & \{a\} \cup (fn(P) - \{\tilde{x}\}) & bn(a\tilde{x}.P) & \stackrel{\text{def}}{=} & bn(P) \cup \{\tilde{x}\} \\
fn((\nu\tilde{x})P) & \stackrel{\text{def}}{=} & fn(P) - \{\tilde{x}\} & bn((\nu\tilde{x})P) & \stackrel{\text{def}}{=} & bn(P) \cup \{\tilde{x}\} \\
fn(P+Q) & \stackrel{\text{def}}{=} & fn(P) \cup fn(Q) & bn(P+Q) & \stackrel{\text{def}}{=} & bn(P) \cup bn(Q) \\
fn(P \mid Q) & \stackrel{\text{def}}{=} & fn(P) \cup fn(Q) & bn(P \mid Q) & \stackrel{\text{def}}{=} & bn(P) \cup bn(Q) \\
fn(!P) & \stackrel{\text{def}}{=} & fn(P) & bn(!P) & \stackrel{\text{def}}{=} & bn(P)
\end{array}$$

Table 2.5: **Free and Bound Names:** Input and restriction bind free names in the continuations of the processes. Therefore, $\tilde{x}$ in the respective rules is subtracted from the free names and added to the bound names. Also, object and subjects from an output prefix are added to the free but not to the bound names. All other rules correspond.

Input-prefix and restriction act as binders. Therefore, the names in their scope are bound; all other names are free. Free and bound names of a process can be computed by standard primitively recursive functions *fn* and *bn*, respectively; see Table 2.5 for a definition. There are two ways of defining whether a name is *fresh*. The more radical way is to require that it does not occur among the names of a process; a more relaxed condition only requires that it does not occur among the free names. The two formulations are equivalent, because bound names are subject to α-conversion. In formalizations it is often convenient to choose names according to the first definition, because it does not necessitate capture-avoiding substitutions when instantiating a process with the fresh name.

When deriving transitions, it is essential to make a clear distinction between free and bound names. Consider the procedure $!ay.\bar{y}c.0$ and an instance $(\nu x)(\bar{a}x.xz.\bar{z}b.0)$ of the caller, yielding the following sequence of transitions:

$$!ay.\bar{y}c.0 \mid (\nu x)(\bar{a}x.xz.\bar{z}b.0) \ \xrightarrow{\tau}\ \xrightarrow{\tau}\ !ay.\bar{y}c.0 \mid (\nu x)(0 \mid \bar{c}b.0) \ \xrightarrow{\bar{c}b}\ !ay.\bar{y}c.0 \mid (\nu x)(0 \mid 0).$$

Now, suppose $x = c$ which so far we have implicitely assumed not to be the case. Then α-equivalence would no longer allow us to release the procedure from the restriction introduced by the call, and the final visible output-transition, $\bar{c}b$, could never take place, because of the subject being local:

$$!ay.\bar{y}c.0 \mid (\nu c)(\bar{a}x.xz.\bar{z}b.0) \ \xrightarrow{\tau}\ \xrightarrow{\tau}\ (\nu c)(!ay.\bar{y}c.0 \mid 0 \mid \bar{c}b.0) \ \xrightarrow{?}\ .$$

In contrast to CHOCS [Tho90], where this kind of *dynamic* binding is used particularly in order to exploit these above effects, the π-calculus adheres to a *static* binding, where α-conversion takes care that previously free names do not become bound by a restriction stemming from a communication like the above procedure-call.

α-equivalence We assume the usual notion of α-equivalence, $\equiv_\alpha$. This allows us to rename bound names in order to avoid capture of free parameters by an extended restriction, as it has happened in the above example. Further, we implicitely assume β-reduction.

$$\frac{}{\tau.P \xrightarrow{\tau} P}\,\mathbf{P1a} \qquad \frac{}{\bar{a}\tilde{b}.P \xrightarrow{\bar{a}\tilde{b}} P}\,\mathbf{P1b} \qquad \frac{\{\tilde{b}\} \cap bn(P) = \emptyset}{a\tilde{x}.P \xrightarrow{a\tilde{b}} P\{\tilde{b}/\tilde{x}\}}\,\mathbf{P1c}$$

$$\frac{P \xrightarrow{\alpha} P' \quad \{\tilde{x}\} \cap fn(\alpha) = \emptyset}{(\nu\tilde{x})P \xrightarrow{\alpha} (\nu\tilde{x})P'}\,\mathbf{P2a} \qquad \frac{P \xrightarrow{\bar{a}(\tilde{b})} P' \quad \{\tilde{x}\} \subseteq \{\tilde{b}\}}{(\nu\tilde{x})P \xrightarrow{(\nu\tilde{x})\bar{a}\tilde{b}} P'}\,\mathbf{P2b} \qquad \frac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'}\,\mathbf{P3a}$$

$$\frac{P \xrightarrow{\alpha} P' \quad bn(\alpha) \cap fn(Q) = \emptyset}{P \,|\, Q \xrightarrow{\alpha} P' \,|\, Q}\,\mathbf{P4a} \qquad \frac{P \xrightarrow{(\nu\tilde{x})\bar{a}\tilde{b}} P' \quad Q \xrightarrow{a\tilde{b}} Q' \quad \{\tilde{x}\} \cap fn(Q) = \emptyset}{P \,|\, Q \xrightarrow{\tau} (\nu\tilde{x})(P' \,|\, Q')}\,\mathbf{P5a}$$

$$\frac{P \xrightarrow{\alpha} P' \quad a = b}{[a = b]P \xrightarrow{\alpha} P'}\,\mathbf{P6} \qquad \frac{P \xrightarrow{\alpha} P' \quad a \neq b}{[a \neq b]P \xrightarrow{\alpha} P'}\,\mathbf{P7} \qquad \frac{!P \,|\, P \xrightarrow{\alpha} P'}{!P \xrightarrow{\alpha} P'}\,\mathbf{P8} \qquad \frac{P \equiv_\alpha P'' \quad P'' \xrightarrow{\alpha} P'}{P \xrightarrow{\alpha} P'}\,\mathbf{P9}$$

Table 2.6: **Operational Semantics of the polyadic π-Calculus.** The *labelled transition-system* for $\mathcal{P}$ is the least set of triples $P \xrightarrow{\alpha} P'$ described by the rules **P1**–**P9**. Rules **P3b**–**P5b** are symmetric versions of **P3a**–**P5a**, so we omit them here.

Operational semantics The behaviour of a process is determined by the names contained in its description, and by the fact whether they are free (or, global) or bound (or, private). As for CCS, we follow an *operational* approach, describing the behaviour of $\mathcal{P}$ in terms of *labelled transition-rules*. An alternative solution often applied is a *reduction-semantics* where only silent transitions are considered (as reductions), and the rest is dealt with by *structural congruence*, reminiscient of an algebraic treatment of process terms. A fully algebraic approach is rarely applied, because there is no complete algebraic description of any of the usual notions of bisimilarity for the π-calculus so far. Again, the *labelled transition-system* is defined *inductively* employing transitions of the form $P \xrightarrow{\alpha} P'$, see Table 2.6. In contrast to the transition-system for CCS, the distinction between free and bound names plays a vital role. For instance, in Rule **P4a** concerning parallel composition, it is explicitely required that in $P \xrightarrow{\alpha} P'$, the local names of P emitted in α do not coincide with the free names of Q put in parallel. This happens in order to guarantee static binding, preventing capturing of free names, as discussed in the above example.

Early and late transitions The operational semantics as defined in Table 2.6 is *early*; that is, the objects received in an input are instantiated immediately. Alternatively, people often consider *late* semantics, in which an instantiation is delayed until a corresponding output. In a late labelled transition-system, the substitution would then occur in Rules **P5a** and **P5b** instead of Rule **P1c**, yielding (we omit Rule **P5b′** which is a symmetric version of Rule **P5a′**),

$$\frac{}{a\tilde{x}.P \xrightarrow{a\tilde{x}} P}\,\mathbf{P1c'} \qquad \frac{P \xrightarrow{(\nu\tilde{x})\bar{a}\tilde{b}} P' \quad Q \xrightarrow{a\tilde{y}} Q' \quad \{\tilde{x}\} \cap fn(Q) = \emptyset \quad \{\tilde{b}\} \cap bn(Q) = \emptyset}{P \,|\, Q \xrightarrow{\tau} (\nu\tilde{x})(P' \,|\, Q'\{\tilde{b}/\tilde{y}\})}\,\mathbf{P5a'}$$

Weak transitions We derive *weak transitions* $P \overset{\alpha}{\Longrightarrow} P'$ for the π-calculus exactly like those for CCS: the transition $P \overset{\epsilon}{\Longrightarrow} P'$ is defined by an arbitrarily large number of silent steps, $P(\xrightarrow{\tau})^*P'$, and for every α, the transition $P \overset{\alpha}{\Longrightarrow} P'$ abbreviates $P \overset{\epsilon}{\Longrightarrow} P_1 \xrightarrow{\alpha} P_2 \overset{\epsilon}{\Longrightarrow} P'$. Again, $\hat{\alpha}$ denotes α for outputs or inputs, and ϵ for $\alpha = \tau$.

With this definition, we can derive rules **Cw1**–**Cw3** from Section 2.1.1 about CCS, also for weak transitions of the π-calculus:

$$\frac{}{P \stackrel{\epsilon}{\Longrightarrow} P}\,\textbf{Cw1} \qquad \frac{P \stackrel{\alpha}{\longrightarrow} P'}{P \stackrel{\alpha}{\Longrightarrow} P'}\,\textbf{Cw2a} \qquad \frac{P \stackrel{\alpha}{\longrightarrow} P'}{P \stackrel{\hat{\alpha}}{\Longrightarrow} P'}\,\textbf{Cw2b} \qquad \frac{P \stackrel{\epsilon}{\Longrightarrow} P_1 \stackrel{\alpha}{\Longrightarrow} P_2 \stackrel{\epsilon}{\Longrightarrow} P'}{P \stackrel{\alpha}{\Longrightarrow} P'}\,\textbf{Cw3}$$

2.2.2 Bisimilarity

Bisimilarity is the predominant notion of behavioural equivalence for the π-calculus. Considering an early notion of the calculus, we define *weak bisimilarity* as follows:

DEFINITION 2.4 (WEAK BISIMILARITY) A relation $\mathcal{R} \subseteq \mathcal{P} \times \mathcal{P}$ is a *weak bisimulation*, if for all $P\mathcal{R}Q$ and all α, the following holds:

(i) if $P \stackrel{\alpha}{\longrightarrow} P'$, then there exists Q' such that $Q \stackrel{\hat{\alpha}}{\Longrightarrow} Q'$ and $P'\mathcal{R}Q'$.

(ii) if $Q \stackrel{\alpha}{\longrightarrow} Q'$, then there exists P' such that $P \stackrel{\hat{\alpha}}{\Longrightarrow} P'$ and $P'\mathcal{R}Q'$.

Weak bisimilarity for the π-calculus is defined as the union of all weak bisimulations; that is, $\approx_\pi \stackrel{\text{def}}{=} \bigcup\{\, \mathcal{R} \mid \mathcal{R} \text{ is a weak bisimulation} \,\}$. Hence, $P \approx_\pi Q$ if there exists a weak bisimulation containing (P, Q).

Early, late, and open semantics The order of the quantifiers in the definition of a bisimilarity determines the discriminating power of the equivalence. For the π-calculus, three notions have been studied: *early*, *late*, and *open* bisimilarities. Early bisimilarity is the coarsest of the three notions, because for each input object, a different answering derivative can be chosen. By contrast, late bisimilarity requires one derivative to be instantiated with all possible objects. Open bisimilarity [San96b] is even more discriminating, employing closing substitutions on the processes in the relation. See [Qua99] for a survey.

Expansion The notion of *expansion* is even more important for the π-calculus than it is for CCS already, because a lot of bisimilarity proofs employ ‘up to’-techniques, and in particular such based on the expansion-preorder.

DEFINITION 2.5 (EXPANSION) A relation $\mathcal{R} \subseteq \mathcal{P} \times \mathcal{P}$ is an *expansion relation*, if for all $P\mathcal{R}Q$ and all α, the following holds:

(i) if $P \stackrel{\alpha}{\longrightarrow} P'$, then there exists Q' such that $Q \stackrel{\hat{\alpha}}{\longrightarrow} Q'$ and $P'\mathcal{R}Q'$.

(ii) if $Q \stackrel{\alpha}{\longrightarrow} Q'$, then there exists P' such that $P \stackrel{\alpha}{\Longrightarrow} P'$ and $P'\mathcal{R}Q'$.

Like for CCS, expansion for the π-calculus is defined as the union of all expansion relations; that is, $\geq_\pi \stackrel{\text{def}}{=} \bigcup\{\, \mathcal{R} \mid \mathcal{R} \text{ is a weak bisimulation} \,\}$. Hence, $P \geq_\pi Q$ if there exists an expansion relation containing (P, Q).

Proof techniques Like in CCS, one can exploit compositionality of weak bisimilarity with respect to parallel composition and restriction. Also, even if not complete, algebraic laws can be fruitfully applied for various purposes, such as,

$P \mid Q \approx_\pi Q \mid P, \quad P \mid (Q \mid R) \approx_\pi (P \mid Q) \mid R,$	rearrangement of processes
$P \mid 0 \approx_\pi P, \quad (\nu\tilde{x})P \approx_\pi P$ if $\{\tilde{x}\} \cap fn(P) = \emptyset,$	garbage collection
$(\nu\tilde{x})(P \mid Q) \approx_\pi (\nu\tilde{x})P \mid Q$ if $\{\tilde{x}\} \cap fn(Q) = \emptyset.$	contraction of restriction

Further, as well like in CCS, expansion can be made use of to reduce the size of the bisimulation relation in an 'up to'-proof.

Definition 2.6 (Bisimilation up to Expansion) A relation $\mathcal{R} \subseteq \mathcal{P} \times \mathcal{P}$ is a *weak bisimulation up to expansion*, if for all $P\mathcal{R}Q$ and all α, the following holds:

(i) if $P \xrightarrow{\alpha} P'$, then there exist P'', Q', Q'' such that $Q \stackrel{\hat{\alpha}}{\Longrightarrow} Q'$ and $P' \geq_\pi P''\mathcal{R}Q'' \leq_\pi Q'$.

(ii) if $Q \xrightarrow{\alpha} Q'$, then there exist P', P'', Q'' such that $P \stackrel{\hat{\alpha}}{\Longrightarrow} P'$ and $P' \geq_\pi P''\mathcal{R}Q'' \leq_\pi Q'$.

In Chapter 4, we make extensive use of bisimilarity proof-techniques, when proving equivalences of Algol-phrases via a translation into the π-calculus.

2.2.3 Type-Systems

The fact that the π-calculus employs only two primitives, makes it a role model for mobile systems, similar to the λ-calculus for sequential computations. Like for the λ-calculus, this generality through simplicity comes with a price to pay: structural information about the systems is completely lost. For instance, (1) it is not clear for a channel a priori, how many names can be sent along it, giving rise to process-terms of the form $\bar{a}b.0 \mid a(x,y).P$, where the prefixes of the sender and receiver have different arities. Or, (2) there can be a need for implicit conventions that certain processes can only read from certain channels, but never write to them. Or, (3) one might like to use certain names to express integer-values, for instance. Or, as a final example, (4) one might assume that certain channels expire after being used a number of times.

These and other requirements can be regarded as properties determining subsets of $\mathcal{P}$ which satisfy them. A popoular approach to computing these subsets is the use of static typing: typable processes belong to the desired set, all others are excluded. In this section, we briefly review type systems proposed for each of the above requirements.

Remark: Note that by reducing the number of processes, type-systems decrease the number of contexts as well. As a result, equivalences become coarser, simply because certain contexts distinguishing between two processes are among those that are eliminated.

Sorting A basic type-system for the polyadic π-calculus was proposed by Milner [Mil91], in order to prevent that the arities of the prefixes of a sender and a receiver differ in a communication, as in our above example, $\bar{a}b.0 \mid a(x,y).P$. According to Milner's *sorting*, each name a is assigned a type (or, *sort*) of the form $\updownarrow [t_1, \dots, t_n]$ giving the arity n and types t_i of the values that a carries in a message. The rules for composing processes say that P and Q can be joined as $P + Q$ or $P \mid Q$ if they yield corresponding types for all their free names.

In our example, name a in $\bar{a}b.0$ has the sort $\updownarrow[t_b]$, given that t_b is the type of b. On the other hand, it has sort $\updownarrow[t_x, t_y]$ in $a(x, y).P$, with t_x and t_y being the types of x and y, respectively. At this point, the sorting tells us that $\bar{a}b.0$ and $a(x, y).P$ cannot match, with a being of contradicting sorts $\updownarrow[t_b]$ and $\updownarrow[t_x, t_y]$; hence $\bar{a}b.0 \mid a(x, y).P$ is not typable and, therefore, does not belong to the admitted set of processes.

I/O-types This type-system can be further refined, for instance, by adding information that in a certain typing-context, a certain name may only be used for reading or writing (or both), as proposed by Odersky [Ode95], and Pierce and Sangiorgi [PS96]. A channel of type $\uparrow[t_1, \ldots, t_n]$ may be used to emit values of types $t_1, \ldots, t_n$, whereas a channel of type $\downarrow[t_1, \ldots, t_n]$ may receive tuples of type $t_1, \ldots, t_n$; a channel of type $\updownarrow[t_1, \ldots, t_n]$ may be used for both.

As an example, consider $\bar{a}b.0$ with type $\uparrow[t_b]$ for name a, as well as a process P. If P can be consistently typed with a being of type $\uparrow[t_b]$ or $\updownarrow[t_b]$, then the process $\bar{a}b.0 \mid ax.P$ is well-typed for a of type $\updownarrow[t_b]$, and belongs to the set of admissible processes; otherwise, it does not.

I/O-types are particularly useful when modelling programming languages with state. Due to Milner [Mil89], a variable x can be modelled by a *register* of the form,

$$\mathbf{Reg}_x[v] \stackrel{\text{def}}{=} \overline{\mathbf{in}}_x v.\mathbf{Reg}_x[v] + \mathbf{out}_x w.\mathbf{Reg}_x[w],$$

where we use agent-notation for the sake of readability; it can be encoded in terms of replication, however, see [Mil91].

Assume a global variable x—or, $\mathbf{Reg}_x$—to be accessed by a program $\mathbf{in}_x y.P$; that is, we have $\mathbf{Reg}_x[v] \mid \mathbf{in}_x y.P$. Without introducing the I/O-types from above, it is not clear that the program has to ask $\mathbf{Reg}_x$ for its value; it could equally well receive a value from the context. An I/O-typing giving read-capability only to $\mathbf{in}_x$ in both program and context, prevents exactly this circumvention of $\mathbf{Reg}_x$. We make use of this type-system in Chapter 4, where we model Concurrent Idealized Algol in the π-calculus. There, we apply registers of exactly the above form to keep track of the states of programs; we will see that there exist programs which necessitate intermediate registers between programs and contexts.

Types for values As pointed out in [Mil91, PS96], basic types of values like booleans or integers can be encoded in the π-calculus. For a simpler usage, it is often convenient to assign to certain names the functionality of a value, in a further simple refinement of the I/O-typing.

Linearity Sometimes, one might wish to express that a name a may only be used a certain number of times, for instance, to model locks preventing a group of processes from entering a critical section unless it is free. A type-system solving this problem using *linear* channels was proposed by Kobayashi, Pierce, and Turner [KPT99].

In Chapter 4, we use a locking-mechanism to implement blocking of the context during the execution of a dedicated **await**-command: whenever the context wants to access a global variable, it has to acquire a lock; the subsequent read- or write-operation is then endowed with a linear type so to be used only once.

2.3 General-Purpose Theorem-Proving

Theorem-provers implement *logical frameworks*, for example, simple type theory [Chu40] like HOL and Isabelle/HOL, the calculus of inductive constructions [Wer94] like Coq, or the LF

logical framework [HHP93] like Elf and its successor Twelf. These environments allow the user to define new objects and deduce proofs about them, either interactively or fully automatically.

General-purpose theorem-provers aim at providing platforms for human-style reasoning which are as general as possible, and therefore apply powerful logics. It is an explicit goal of these tools to offer the user human-style strategies for reasoning. As a consequence, the user can formulate definitions and proofs in a natural way, but cannot expect full automation and efficiency in the deduction of proofs. Some of the most influential general-purpose interactive theorem-provers are Coq [BBC+99], HOL [GM93], Isabelle [Pau94], and PVS [SOR93].

Special-purpose theorem-provers are usually based on first-order deduction-systems, and derive proofs fully automatically. This does not mean, however, that the user can insert an arbitrary problem; instead, he/she has to prepare a series of small proof-steps which the prover can solve automatically, and then combine them in a final theorem. Often the order in which the single proofs are arranged plays a vital role for a successful completion of the whole proof. This necessitates a good understanding of the underlying proof-system on part of the user. Special-purpose theorem-provers are usually designed to infer proofs efficiently, and to tackle even large proofs. One area of application is hardware verification; for instance, the AMD5K86 has been verified in the Boyer-Moore prover ACL2 [KMM00b, KMM00a], see [BKM96].

More recently, *higher-order logical frameworks* have gained more importance, for example, ELF [Pfe89] and its successor Twelf [PS99], or λProlog [NM98]. These frameworks are especially designed for the formalization of languages with binders, such as high-level programming languages. By delegating the treatment of bound variables to the prover, they save the user the from effort of implementing and applying substitutions for α-conversion and β-reduction.

2.3.1 Isabelle/HOL

We use the interactive theorem-prover Isabelle [Pau94] in its instantiation for higher-order logic [Pau93]. The environment is similar to the HOL system [GM93]; both are extensions of Church's simple type theory [Chu40] adding, for instance, polymorphism and type-classes. In this section, we give an overview of the main features of Isabelle/HOL, and show how they can be applied in the analysis of processes.

Genericity Isabelle [Pau94] is designed as a *generic* theorem-prover, and therefore clearly distinguishes between its *meta-level* and *object-level*. The meta-level can be thought of as the 'hardware' of the prover: it consists of an intentionally small kernel based on intuitionistic higher-order logic, and is provided by the implementors. Upon it, on the object-level, *object-logics* [Pau93] can be formally derived in terms of definitions and proofs. Isabelle then guarantees for the correctness of this 'software', modulo the correctness of the small kernel. The intention behind this generic approach is to guarantee both a maximum of flexibility and a maximal degree of correctness.

Isabelle's object-logics Various object-logics come with the distribution of Isabelle [Pau93], such as first-order logic, (Isabelle/FOL), constructive type theory (Isabelle/CTT), or higher-order logic (Isabelle/HOL). We use higher-order logic (HOL), because it allows for a human-style way of reasoning, and it already provides a range of datatypes like sets or lists, and results proved about them. Further, a range of powerful proof-procedures have been designed for Isabelle/HOL.

The object-logic HOL is implemented in terms of a set of *theories*, which are brought together in `Main.thy`. When applying Isabelle/HOL, users normally include this main theory as a basis for further extensions.

Theories The user can introduce *theories* on top of an object-logic, defining new objects and deriving proofs. As an example, assume a formalization of the finite subset of pure CCS without relabelling (see Section 2.1.1), with processes of the form,

$$P \ ::= \ 0 \ | \ \alpha.P \ | \ P\backslash L \ | \ P+P \ | \ P\,|\,P.$$

Our formalization consists of three theories: one defining the syntax (`CCS_Syntax`), one giving the semantics (`CCS_Semantic`), and one introducing observation equivalence (`CCS_Equiv`). Each of the theories consists of two parts: in a `CCS_xxx.thy`-file, we give the basic definitions of syntax and semantics, whereas in a `CCS_xxx.ML`-file, we derive corresponding theorems. The proof-scripts are available at `http://www7.in.tum.de/~roeckl/thesis/tiny/`.

Defining theories Consider the theory-file `CCS_Syntax.thy`, as depicted in Table 2.7. In its header, the theory `Main` is included in order to access the full functionality of Isabelle/HOL. The datatype-definitions for labels (`'a labels`), actions (`'a actions`), and processes (`'a procs`) consist of a BNF-like notation equipped with the types of the single components. The types are parameterized over `'a` denoting the set of names, and for which arbitrary types can be instantiated later. This allows us to defer the decision about a concrete set of names. The datatype-definitions consist of a BNF-like notation augmented with the types of the arguments: a prefixed process $\alpha.P$, for example, takes an action α and a process P as arguments, and is therefore of type `('a actions)` $\times$ `('a procs)` $\rightarrow$ `('a procs)`. Datatype-definitions in Isabelle/HOL automatically yield principles for structural induction and case-distinction. We make use of related principles in large style in Chapter 3, where we formalize the syntax of the π-calculus.

Isabelle allows the user to give a *concrete syntax* to datatypes. In `CCS_Syntax.thy`, we do so for the processes. For an action prefix $\alpha.P$, we can then write `[alpha].P` instead of `Prefix alpha P`; similarly for the other constructs. Isabelle takes care of correct translations between *concrete* (`[alpha].P`) and *abstract* (`Prefix alpha P`) syntax automatically. Note that this feature of Isabelle considerably enhances readability, especially in large proofs.

Modularity Larger theories in Isabelle should be modularized into several sub-theories; in our case, we have split the formalization into three parts. The theory `CCS_Semantic`, for instance, builds on the theory-file `CCS_Syntax.thy`—which it includes in its header—and implements a labelled transition-system for `'a procs`; see Table 2.8. In the first part of the file, we declare constants for strong and weak transitions (`consts`), and specify a concrete syntax for them (`syntax` and `translations`). Then, in a second part, we define the strong transition-rules (`inductive StTrans`) and give rules for the derivation of weak transitions.

Inductive sets We specify the transition-system for CCS by giving a set of *introduction-rules* and declaring it to be *inductive*. Note that the rules in Table 2.8 are simple transcriptions of the corresponding rules in Table 2.2. As pointed out in Section 2.1.1, inductiveness means that the set of transitions consists exactly of those triples $P \stackrel{\alpha}{\longrightarrow} P'$ that can be derived from

```
CCS_Syntax = Main +

datatype
  'a labels  = nm    'a
             | conm  'a

datatype
  'a actions = tau
             | va   ('a labels)

datatype
  'a procs =
     Nil                                               (".0" 115)
   | Prefix         ('a actions) ('a procs)            ("[_]._" [120,110] 110)
   | Restriction    ('a procs) (('a labels) set)       ("_.[_]" [100,120] 100)
   | Choice         ('a procs) ('a procs)              (infixl ".+" 80)
   | Composition    ('a procs) ('a procs)              (infixl ".|" 90)

constdefs
  compl :: "'a labels => 'a labels"
 "compl l == case l of nm a => nm a | conm a => conm a"

end
```

Table 2.7: **Formalizing the Syntax of CCS:** We implement the syntax of the finite subset of pure CCS in a theory `CCS_Syntax.thy`. Labels, actions, and processes are formalized in *datatype-definitions*, which automatically yield principles for structural induction and case-distinction. For example, Isabelle can automatically deduce that a visible action (that is, a label) cannot equal the silent action τ. We use the symmetric function `compl` to determine the complement of a label.

```
CCS_Semantic = CCS_Syntax +

consts
  StTrans :: "('a procs * 'a actions * 'a procs) set"
  WkTrans :: "('a procs * 'a actions * 'a procs) set"
  WkEps   :: "('a procs * 'a procs) set"
  ...

syntax
 "@StTrans" :: ['a procs, 'a actions, 'a procs] => bool
              ("_/ -(_)->/ (7_)" [80, 0, 80] 70)
 ...

translations
 "P -alpha-> P'"  == "(P, alpha, P') : StTrans"
 ...

inductive StTrans
  intrs
  C1  "[alpha].P -alpha-> P"
  C2a "P -tau-> P' ==> P.[L] -tau-> P'.[L]"
  C2b "[| P -va l-> P' ; l ~: L ; compl l ~: L |] ==> P.[L] -va l-> P'.[L]"
  C3a "P -alpha-> P' ==> P .+ Q -alpha-> P'"
  C3b "Q -alpha-> Q' ==> P .+ Q -alpha-> Q'"
  C4a "P -alpha-> P' ==> P .| Q -alpha-> P' .| Q"
  C4b "Q -alpha-> Q' ==> P .| Q -alpha-> P .| Q'"
  C5  "[| P -va l-> P' ; Q -va (compl l)-> Q' |] ==> P .| Q -alpha-> P' .| Q'"

defs
  WkEps_def   "WkEps == {(P, P') . P -tau-> P'}^*"
  WkTrans_def "WkTrans == {(P, alpha, P') . EX P1 P2. \
              \ P =eps=> P1 & P1 -alpha-> P2 & P2 =eps=> P'}"
  ...

end
```

Table 2.8: **Formalizing the Semantic of CCS:** A labelled transition-system is formalized in terms of an inductive set of rules for `StTrans`. A concrete syntax allows us to use arrow-notation. Weak transitions are derived according to the usual definition applying Kleene's star on silent transitions; see Section 2.1.1.

the set of transition-rules. From this definition, Isabelle automatically computes corresponding principles for rule-induction and case-exhaustion that can be applied in subsequent proofs.

Deriving theorems As mentioned above, a theory usually consists of two parts: in a `.thy`-file, objects are defined, whereas in a corresponding `.ML`-file, theorems are derived for these objects. The file itself contains the proof-scripts specifying the interaction with Isabelle during a proof. As an example, consider the proof-scripts in Table 2.9, in which we derive laws **Cw1**–**Cw3** from Section 2.1.1. They are collected in a `CCS_Semantic.ML`-file, from which Isabelle automatically concludes that they belong to the theory `CCS_Semantic`; hence it loads them together with `CCS_Semantic.thy`.

As a typical example, consider the proof of law **Cw2b**, stating that every strong transition gives rise to a corresponding weak transition with a hat (recall that $\hat{\alpha} = \alpha$ if α is visible, and $\hat{\tau} = \epsilon$):

```
Goal "P -alpha-> P' ==> P =^alpha=> P'";
by (case_tac "alpha" 1);
by (auto_tac (claset() addIs [Cw1a,Cw2a],
              simpset() addsimps [WHTrans_def]));
qed "Cw2b";
```

The derivation consists of three parts: first, the goal is stated in concrete syntax (`Goal`); then the proof is derived applying Isabelle's tactics (`by`); and, once it is completed, the goal is stored in Isabelle's database as a theorem (`qed`) called `Cw2b`. The proof itself is derived interactively with Isabelle in a backward-resolution style. After stating the goal,

```
> Goal "P -alpha-> P' ==> P =^alpha=> P'";
Level 0 (1 subgoal)
P -alpha-> P' ⟹ P =^alpha=> P'
 1. P -alpha-> P' ⟹ P =^alpha=> P'
val it = [] : thm list,
```

we ask Isabelle to make a case-distinction on action `alpha`, whether it is silent (`tau`), or is a visible action consisting of a label (`va labels`), which yields two respective subgoals:

```
> by (case_tac "alpha" 1);
Level 1 (2 subgoals)
P -alpha-> P' ⟹ P =^alpha=> P'
 1. ⟦ P -alpha-> P' ; alpha = tau ⟧ ⟹ P =^alpha=> P'
 2. ⋀ labels. ⟦ P -alpha-> P' ; alpha = va labels ⟧ ⟹ P =^alpha=> P'
val it = () : unit,
```

Isabelle is able to prove both of them fully automatically, by one application of its automatic tactic `auto_tac`:

```
> by (auto_tac (claset() addIs [Cw1a,Cw2a],
                simpset() addsimps [WHTrans_def]));
Level 2
P -alpha-> P' ⟹ P =^alpha=> P'
No subgoals!
val it = () : unit,
```

```
Goal "P =eps=> P";
by (simp_tac (simpset() addsimps [WkEps_def]) 1);
qed "Cw1";

Goal "P -tau-> P' ==> P =eps=> P'";
by (auto_tac (claset() addIs [r_into_rtrancl],
              simpset() addsimps [WkEps_def]));
qed "Cw1a";

Goal "P -alpha-> P' ==> P =alpha=> P'";
by (auto_tac (claset() addSIs [Cw1], simpset() addsimps [WkTrans_def]));
qed "Cw2a";

Goal "P -alpha-> P' ==> P =^alpha=> P'";
by (case_tac "alpha" 1);
by (auto_tac (claset() addIs [Cw1a,Cw2a],
              simpset() addsimps [WHTrans_def]));
qed "Cw2b";

Goalw [WkEps_def] "[| P =eps=> P1 ; P1 =eps=> P' |] ==> P =eps=> P'";
by (dtac rtrancl_trans 1);
by (Auto_tac);
qed "Cw3a";

Goal "P =tau=> P' ==> P =eps=> P'";
by (auto_tac (claset() addSDs [Cw1a] addIs [Cw3a],
              simpset() addsimps [WkTrans_def]));
qed "Cw3b";

Goal "[| P =eps=> P1 ; P1 =alpha=> P2 ; P2 =eps=> P' |] ==> P =alpha=> P'";
by (auto_tac (claset() addSIs [Cw3a],
              simpset() addsimps [WkTrans_def]));
qed "Cw3";
```

Table 2.9: **Semantic Results about CCS:** Results about strong and weak transitions are derived. Most of them follow easily from the definitions in `CCS_Semantic.thy` by applying an automatic tactic. In particular, we prove laws **Cw1**–**Cw3** from Section 2.1.1.

```
constdefs
  is_WkB :: "('a procs * 'a procs) set => bool"
 "is_WkB R == ALL P Q alpha P' Q' . (P, Q) : R --> \
           \ (P -alpha-> P' --> (EX Q' . Q =^alpha=> Q' & (P', Q') : R)) & \
           \ (Q -alpha-> Q' --> (EX P' . P =^alpha=> P' & (P', Q') : R))"

  WkB :: "('a procs * 'a procs) set"
 "WkB == Union {R . is_WkB R}"
```

Table 2.10: **Formalizing Observation-Equivalence:** We define the question whether a relation is a bisimulation in terms of a predicate `is_WkB`. Observation-equivalence, `WkB`, is then formalized as the union over all relations satisfying `is_WkB`.

The tactic employs the predefined sets `claset()` and `simpset()`, the former containing classical rules of the form $[\![P_1; \dots ; P_n]\!] \Longrightarrow P$ for $n \geq 0$ and the latter containing equations for algebraic transformations. Both can be augmented by the user. In our example, we add `Cw1a` and `Cw2a` (see Table 2.9) to `claset()`, and `WHTrans_def` (weak transitions with a hat) to `simpset()`. With no subgoal being left over, we can store the theorem in Isabelle's database:

```
> qed "Cw2b";
val Cw2b = "P -alpha-> P' ==> P =^alpha=> P'" : thm
val it = () : unit
```

In the proof, we have equipped `auto_tac` with the theorems `Cw1a` and `Cw2a` derived previously—see Table 2.9 for their proofs—as well as with the definition of transitions with a hat.

Predicates We specify the question whether a relation is a bisimulation or not, in terms of a predicate `is_WkB`; see Table 2.10 for a formalization. The two proof-obligations can be formalized in a natural way. According to Definiton 2.1, observation-equivalence is the union of all bisimulations, which we can express in Isabelle by using the `Union`-constructor from its theory for sets, `Set.thy`.

Tactics and tacticals Isabelle offers two principal kinds of tactics: *classical* tactics based on resolution, and *algebraic* tactics applying term-rewriting. Classical tactics, like `resolve_tac` for backward-resolution or `forward_tac` for forward-resolution, refine subgoals by unifying them with specified parts of a definition or theorem. Algebraic tactics, like `simp_tac`, try to solve subgoals by replacing expressions within them by simpler ones. They operate with theorems that do not have premises; or, if they have, only very simple ones. Note that in Isabelle there is no distinction between definitions and theorems concerning their application in proofs. As an example of the use of `forward_tac`, consider the proof that observation-equivalence is a bisimulation (Theorem `WkB_is_WkB` in Table 2.11). There, two subgoals result from an application of the definition of observation-equivalence, one for P making a transition in $(P, Q) \in R$, and the other for Q; we will see below how they can be suitably derived in Isabelle/HOL. The one for P is of the following form:

1. $[\![\, (P, Q) \in R;\ \texttt{is_WkB}\ R;\ P \xrightarrow{\alpha} P' \,]\!]$
   $\Longrightarrow\ \exists Q'.\ Q \overset{\hat{\alpha}}{\Longrightarrow} Q' \ \wedge\ (\exists X.\texttt{is_WkB}\ X \wedge (P', Q') \in X)$

```
Goalw [is_WkB_def]
  "ALL P Q alpha P' Q' . (P, Q) : R --> \
    \ (P -alpha-> P' --> (EX Q' . Q =^alpha=> Q' & (P', Q') : R)) & \
    \ (Q -alpha-> Q' --> (EX P' . P =^alpha=> P' & (P', Q') : R)) \
  \ ==> is_WkB R";
by (Fast_tac 1);
qed "is_WkB_I";

Goalw [is_WkB_def] "is_WkB R ==> \
 \ ALL P Q alpha P' Q' . (P, Q) : R --> \
    \ (P -alpha-> P' --> (EX Q' . Q =^alpha=> Q' & (P', Q') : R)) & \
    \ (Q -alpha-> Q' --> (EX P' . P =^alpha=> P' & (P', Q') : R))";
by (Fast_tac 1);
qed "is_WkB_D";

Goal "[| is_WkB R ; (P, Q) : R ; P -alpha-> P' |] \
 \ ==> EX Q' . Q =^alpha=> Q' & (P', Q') : R";
by (auto_tac (claset() addSDs [is_WkB_D], simpset()));
qed "is_WkB_D1";

Goal "[| is_WkB R ; (P, Q) : R ; Q -alpha-> Q' |] \
 \ ==> EX P' . P =^alpha=> P' & (P', Q') : R";
by (auto_tac (claset() addSDs [is_WkB_D], simpset()));
qed "is_WkB_D2";

Goalw [WkB_def] "is_WkB WkB";
by (auto_tac (claset() addSIs [is_WkB_I], simpset()));
by ((forward_tac [is_WkB_D1] 1) THEN (REPEAT (atac 1)));
by ((forward_tac [is_WkB_D2] 2) THEN (REPEAT (atac 2)));
by (Auto_tac);
qed "WkB_is_WkB";
```

Table 2.11: **Results about Observation-Equivalence:** We derive introduction- and destruction-rules for weak bisimulations, by applying Isabelle's automatic tactics. Finally, we prove that observation-equivalence is itself a bisimulation (`WkB_is_WkB`). Here, we explicitely apply Isabelle's classical tactic `forward_tac`.

stating that for arbitrary pairs $(P, Q) \in R$ such that R is a weak bisimulation and $P \xrightarrow{\alpha} P'$ (premises), there exists a suitable Q' such that $Q \stackrel{\hat{\alpha}}{\Longrightarrow} Q'$ and $(P', Q') \in X$ for some weak bisimulation X. Note that Isabelle/HOL does not require X to coincide with R. The existential quantification stems from the union over all weak bisimulations in the definition of observation-equivalence; compare Definition 2.1. By applying the destruction-rule `is_WkB_D1` from Table 2.11, mapping P and Q to the process-identifiers there, we obtain a suiting transition of Q such that even $(P', Q') \in R$:

$$1.\ [\![\,(P,Q) \in R;\ \ \texttt{is_WkB}\ R;\ \ P \xrightarrow{\alpha} P';\ \ \exists Q'.\, Q \stackrel{\hat{\alpha}}{\Longrightarrow} Q' \wedge (P',Q') \in R\,]\!]$$
$$\Longrightarrow\ \ \exists Q'.\ \ Q \stackrel{\hat{\alpha}}{\Longrightarrow} Q'\ \wedge\ (\exists X.\,\texttt{is_WkB}\ X \wedge (P',Q') \in X)$$

The suitable instantiations of the premises `(P, Q) : R` and `P -alpha-> P'` can be removed by assumption, using `assume_tac`—or `atac` for short—repeatedly. The respective part of the interaction with Isabelle is as follows (for the sake of readability, we use sugared notation):

$$1.\ [\![\,(P,Q) \in R;\ \ \texttt{is_WkB}\ R;\ \ P \xrightarrow{\alpha} P'\,]\!]$$
$$\Longrightarrow\ \ \exists Q'.\ \ Q \stackrel{\hat{\alpha}}{\Longrightarrow} Q'\ \wedge\ (\exists X.\texttt{is_WkB}\ X \wedge (P',Q') \in X)$$

...

```
> by ((forward_tac [is_WkB_D1] 1) THEN (REPEAT (atac 1)));
```

$$1.\ [\![\,(P,Q) \in R;\ \ \texttt{is_WkB}\ R;\ \ P \xrightarrow{\alpha} P';\ \ \exists Q'.\, Q \stackrel{\hat{\alpha}}{\Longrightarrow} Q' \wedge (P',Q') \in R\,]\!]$$
$$\Longrightarrow\ \ \exists Q'.\ \ Q \stackrel{\hat{\alpha}}{\Longrightarrow} Q'\ \wedge\ (\exists X.\,\texttt{is_WkB}\ X \wedge (P',Q') \in X)$$

...

In the above proof-step, we have combined two tactics by using Isabelle's *tacticals*. With these basic operators, the user can build more elaborate tactics from already existing ones. Isabelle's automatic tactics are an example of such elaborate constructions. In Chapters 3 and 5, we make particular use of `auto_tac` and `force_tac`, which both combine classical and algebraic reasoning. To return to our example, we apply `auto_tac` both to unfold the definition of observation-equivalence in the first proof step and to instantiate identifier X with R in the last one. As is obvious from Tables 2.9 and 2.11, the automatic tactics apply predefined classical (`claset()`) and algebraic (`simpset()`) sets of rules. The classical set `claset()` contains introduction and elimination rules; the latter are a special formulation of destruction rules. The simplification set `simpset()` contains equations to be used in algebraic transformations. The two sets can be augmented by the user by inserting or deleting specified rules. The set `claset() addSIs [is_WkB_D1]` in our above example, for instance, augments the classical set with the destruction-rule `is_WkB_D1`.

2.3.2 Representing Languages with Binders

In the previous section, we have presented a formalization of finitary pure CCS in Isabelle/HOL. More elaborate languages, such as the π-calculus, usually employ *binders*; that is, some of their operators abstract over a part of the continuation. In the π-calculus, there are two binders: input $ax.P$ and restriction $(\nu x)P$. Generally, languages with binders can be represented either in *first-order syntax* or *higher-order (abstract) syntax (HOAS)*. First-order syntax is the classical

First-order syntax

P_f ::=	0	Inaction	$\mathcal{P}$
|	$\mathbf{\bar{a}b}.P_f$	Output Prefix	$\mathcal{N} \longrightarrow \mathcal{N} \longrightarrow \mathcal{P} \longrightarrow \mathcal{P}$
|	$\mathbf{ab}.P_f$	Input Prefix	$\mathcal{N} \longrightarrow \mathcal{N} \longrightarrow \mathcal{P} \longrightarrow \mathcal{P}$
|	$P_f \mid P_f$	Parallel Composition	$\mathcal{P} \longrightarrow \mathcal{P} \longrightarrow \mathcal{P}$

Higher-order syntax

P_h ::=	0	Inaction	$\mathcal{P}$
|	$\mathbf{\bar{a}b}.P_h$	Output Prefix	$\mathcal{N} \longrightarrow \mathcal{N} \longrightarrow \mathcal{P} \longrightarrow \mathcal{P}$
|	$\mathbf{a}b.f_P(b)$	Input Prefix	$\mathcal{N} \longrightarrow (\mathcal{N} \longrightarrow \mathcal{P}) \longrightarrow \mathcal{P}$
|	$P_h \mid P_h$	Parallel Composition	$\mathcal{P} \longrightarrow \mathcal{P} \longrightarrow \mathcal{P}$

Table 2.12: **First-order and higher-order syntax.** We consider a small fragment $\mathcal{P}$ of the π-calculus. Communication-channels and values are identified as names of type $\mathcal{N}$. The two syntaxes differ with respect to the *input-prefix* which is defined over two names and a process in first-order, but one name and a function from names to processes in higher-order syntax.

way: In the definition of a datatype $\mathcal{T}$ with a BNF equation $T ::= \ldots$, the arguments on the right-hand side employ only $\mathcal{T}$ and previously defined types. In a first-order syntax for the π-calculus, an input $ab.P$ is represented by two objects of type $\mathcal{N}$ for names and an object of type $\mathcal{P}$. In higher-order syntax, the continuations of binders are represented by functions, and bound names are therefore arguments of these functions. In a higher-order syntax for the π-calculus, an input is therefore represented by one object of type $\mathcal{N}$ and an object of type $\mathcal{N} \to \mathcal{P}$.

As an example, let us consider a fragment $\mathcal{P}$ of a π-calculus-style language, with inaction, output, input, and parallel composition:

$$P \quad ::= \quad 0 \ \mid \ \bar{a}b.P \ \mid \ ab.P \ \mid \ P \mid P.$$

A first-order and a higher-order syntax for this calculus are given in Table 2.12. As a constant, inaction implements a type $\mathcal{P}$ in both representations, output yields $\mathcal{N} \to \mathcal{N} \to \mathcal{P} \to \mathcal{P}$, and parallel composition yields $\mathcal{P} \to \mathcal{P} \to \mathcal{P}$. The difference occurs for input, where first-order syntax yields $\mathcal{N} \to \mathcal{N} \to \mathcal{P} \to \mathcal{P}$ like output, and higher-order syntax yields $\mathcal{N} \to (\mathcal{N} \to \mathcal{P}) \to \mathcal{P}$. In abuse of notation, we write $\mathbf{ab}.P$ and $\mathbf{a}b.f_P(b)$, where P corresponds to $f_P(b)$. The parameter b is bound by a λ-abstraction over f_P.

Instantiating processes Consider the process $\bar{a}b.P \mid ax.Q$, where the first process intends to send b to the second one along a in a communication,

$$\bar{a}b.P \mid ax.Q \quad \stackrel{\tau}{\longrightarrow} \quad P \mid Q\{b/x\}.$$

The process $Q\{b/x\}$ originates from Q by replacing every occurrence of x with b. This *instantiation* can be interpreted as a capture-avoiding substitution or as a β-reduction. In arguments about the π-calculus, it is often not clear—and is of no importance, anyway—which of the two techniques is used to implement instantiations. Substitutions are used in first-order syntax, wheras β-reduction is applied in HOAS.

In the sequel, we use $\mathbf{a}, \mathbf{b}, \ldots$ to represent names in first-order syntax and free names in higher-order syntax, and $x, y, \ldots$ for bound names in higher-order syntax. Further, we use a function f_Q to represent Q in higher-order syntax. In first- and higher-order syntax, the above communication is therefore implemented by,

$$\bar{\mathbf{a}}\mathbf{b}.P \mid \mathbf{a}\mathbf{x}.Q \quad \xrightarrow{\tau} \quad P \mid Q\{\mathbf{b}/\mathbf{x}\} \qquad \text{and} \qquad \bar{\mathbf{a}}\mathbf{b}.P \mid \mathbf{a}x.f_Q(x) \quad \xrightarrow{\tau} \quad P \mid f_Q(\mathbf{b}),$$

respectively. Now, suppose that $Q = \mathbf{c}\mathbf{b}.Q'$, employing another input on $\mathbf{b}$ to make it local within Q' in first-order syntax. In this case, the communication does not yield one but at least two substitutions,

$$\bar{\mathbf{a}}\mathbf{b}.P \mid \mathbf{a}\mathbf{x}.\mathbf{c}\mathbf{b}.Q' \quad \xrightarrow{\tau} \quad P \mid \mathbf{c}\mathbf{b}'.Q'\{\mathbf{b}'/\mathbf{b}\}\{\mathbf{b}/\mathbf{x}\},$$

first renaming the $\mathbf{b}$ bound by restriction into some fresh $\mathbf{b}'$, before introducing the received name $\mathbf{b}$ into the process. Using the functional representation f_Q of Q, on the other hand, we obtain a process abstraction $f_Q = \lambda x.\, \mathbf{c}b.ff_{Q'}(b, x)$, in which the variable b is a priori distinct from any other name, hence also from $\mathbf{b}$, according to the functional mechanism.

$$\bar{\mathbf{a}}\mathbf{b}.P \mid \mathbf{a}x.(\nu b)ff_Q(b, x) \quad \xrightarrow{\tau} \quad P \mid (\nu b)ff_Q(b, \mathbf{b}).$$

Of course, in this second variant, α-conversion and β-reduction are not completely absent, only the user does not have to care for them, because they are made transparent by the underlying functional mechanism that implements them. This delegation is particularly useful when dealing with large processes containing a vast number of binders, so a manual treatment of them can be avoided this way. Note that the formalization of capture-avoiding substitutions in theorem-provers is intricate, and errors are likely. Further, substitutions in first-order syntax make a semantic analysis of processes hard.

Structural induction On the other hand, first-order syntax provides structural induction whereas higher-order syntax does not by itself. The reason for this lack of induction in HOAS is the use of functions to represent continuations of binders, whereas induction requires first-order objects. Structural induction is important in structural syntax-analysis, which is again important in the derivation of principles underlying a semantic analysis. It is a prevailing issue to develop suitable induction-principles for HOAS and use them in syntax-analysis. Chapter 3 is devoted to this problem. There we introduce inductive predicates inspired by [DH94, DFH95] describing the class of processes we are interested in, and use the rule-induction offered by these predicates to mimic structural induction.

Deep and shallow embeddings General-purpose theorem-provers distinguish between a *meta-level* and an *object-level*. While the meta-level is provided by the implementation and is—apart from one exception that we discuss below—not accessible to the user, new objects are defined and proofs are derived on the object-level. Isabelle implements a λ-calculus on its meta-level, using a deBruijn-mechanism [deB72] for the treatment of its variables. Free variables in this calculus, $\mathbf{x}, \mathbf{y}, \ldots$, are also called *object-variables*, and bound variables, $x, y, \ldots$, are called *meta-variables*. This λ-calculus on Isabelle's meta-level has recently been made available for use on the object-logic HOL [BW99]. This little exception from the usual encapsulation of the meta-level allows users to apply functions in datatype-definitions, giving way to a direct implementation of HOAS.

```
axclass inf_class < term
  inf_class "EX (f::nat=>'a). inj f"

datatype
  'a procs = Null                                ".0" 115)
           | Out     'a 'a ('a procs)           ("_<_>._" [120, 0, 110] 110)
           | In      'a "'a => ('a procs)"      ("_[_]._" [120, 0, 110] 110)
           | Par     ('a procs) ('a procs)      (infixl ".|" 90)
```

Table 2.13: **A Shallow Embedding:** We implement our example calculus $\mathcal{P}$ in Isabelle/HOL. By using the axiomatic type-class `inf_class`, we make sure that only types can be used for names in processes which are at least countably infinite.

Theorem-provers like Isabelle or Coq offer two alternative ways of formalizing languages with binders: besides a *deep embedding* residing fully within the object-level, there is the possibility of giving a *shallow embedding* residing within the object-level but using the binding-mechanism of the meta-level. First-order syntax is always formalized in a deep embedding, whereas higher-order syntax can be modelled in both deep and a shallow embeddings: according to the first variant, a λ-calculus is formalized in a deep embedding in order to provide the functional mechanism for HOAS, whereas according to the second variant, the λ-calculus from the meta-level is employed. Table 2.13 shows a formalization of $\mathcal{P}$ in a shallow embedding in Isabelle/HOL.

Exotic terms A general problem of shallow embeddings is the presence of *exotic terms* if there are operators on the object-level that can be used in fuction-definitions, such as conditionals. As an example, consider the functions from names to processes,

$$\begin{aligned} f_E &= \lambda x.\ \text{if } \mathbf{a} = x \text{ then } 0 \text{ else } \mathbf{a}b.0, \\ f_W &= \lambda x.\ \mathbf{a}b.0. \end{aligned}$$

The function f_W can be considered as *well-formed* or *valid*, because it is derived entirely from the syntactic description of the language in Table 2.12. The process-abstraction f_E, on the other hand, is an exotic term, because it contains a conditional expression which is not part of the syntax of the language. Correspondingly, the process $\mathbf{b}x.f_W(x)$ is well-formed, whereas $\mathbf{b}x.f_E(x)$ is not.

Higher-order frameworks The distinction between well-formed (that is, valid) and exotic (that is, invalid) terms is important when reasoning about processes in HOAS. The reason is that certain structural properties that obviously hold for well-formed processes, are inconsistent for exotic terms. Such properties naturally have to be stated as *hypothetical judgements* of the form 'if t is a well-formed term, then property P holds for t'. In our formalization in Chapter 3, we use this technique on the object-level by introducing well-formedness predicates. As a consequence, semantic reasoning always involves a treatment of syntactic well-formedness assumptions.

To facilitate proofs for the users, well-formedness assumptions are part of the meta-level of specialized higher-order frameworks, such as Elf [Pfe89] and its successor Twelf [PS99], or

λProlog [NM98]. Under certain circumstances, also Coq [BBC$^+$99] can be used as such [HMS00]. These frameworks possess meta-logics but offer virtually no object-logics, so that in a formalization there are no constructors except for those given explicitely by the user in syntax-declarations. As a consequence, there is no means of defining exotic terms. Yet, neither of these environments allows for structural syntax-analysis, lacking structural induction. However, there are recent attempts to add structural induction to Twelf [DPS97].

Datatypes As usual in programming-language theory, the structure of a process can be described in terms of a *process-tree*: the leaves are marked with a name or with the inaction process, and inner nodes are marked with an operator. The first-order process $\bar{\mathbf{a}}\mathbf{b}.0 \mid \mathbf{a}\mathbf{x}.\bar{\mathbf{x}}\mathbf{c}.0$ and its higher-order counterpart $\bar{\mathbf{a}}\mathbf{b}.0 \mid \mathbf{a}x.\bar{x}\mathbf{c}.0$, for instance, yield trees,

where the right subtree of the higher-order process is essentially a function from names to process-trees. The tree-like structure is directly derivable from the datatype-definitions, and the abstraction in the subtrees of the shallow embedding stems from the functions from names to processes in the datatype definitions for binders.

We exploit the recursiveness of the datatype-definitions when setting up functions for the processes, such as the computation of the free names or the depth of binders. In the shallow embedding, suitable instantiations have to be chosen for the contiuations of binders. For example, when computing the free names, we use a universal quantification over all instantiations, whereas for the depth of binders, we content ourselves with a single instantiation.

Chapter 3

Formalizing the π-Calculus in HOAS

In Chapter 2, we have introduced the π-calculus as a model for mobile systems [MPW92, Mil99]. As a value-passing calculus with global and local names, it is particularly characterized by its binders *input* and *restriction*; with both constructors operating on names, processes can change their topological structure during execution. In this chapter, we develop a straightforward deep and a shallow embedding of the calculus in Isabelle/HOL (Section 3.1). We show how to remedy the two typical problems of higher-order abstract syntax—lack of structural induction and appearance of exotic terms—by means of well-formedness predicates. Then, we discuss induction-techniques for the resulting hypothetical judgements, and use them in the derivation of the theory of contexts for the π-calculus (Section 3.2). Finally, we show within Isabelle/HOL that the shallow embedding is fully adequate with respect to our deep embedding (Section 3.3). We do not develop the deep embedding further than up to α-equivalence and normalization, omitting full β-reduction or α-conversion, because we concentrate on the shallow embedding, and use the deep one only for the adequacy proof.

From a π-calculus point of view, the formalization presented in this chapter provides a platform for the semantic study of the calculus in Isabelle/HOL in which the user is freed from the manual application of substitutions when deriving transitions or bisimulation-proofs. From a logical point of view, the formalization is an exercise in syntax-analysis in HOAS. Well-formedness assumptions that theorem-provers like Elf [Pfe89] or λProlog [NM98] hide on their meta-level, are available on the object-level of our formalization. This enables us in particular to mechanize proofs about HOAS, like the derivations of the theory of contexts and adequacy, which have to be performed on paper for these frameworks. Using inductive well-formedness predicates, we replace usual structural induction—which does not work in HOAS—by rule-induction. From a theorem-proving point of view, this chapter describes a case-study in giving shallow embeddings to languages with binders.

Proof-techniques for HOAS Higher-order abstract syntax does not provide suitable principles for structural induction, hence does not directly allow to derive meta-theoretical results based on syntax-analysis. Despeyroux, Felty, and Hirschowitz propose to mimic structural induction by rule-induction over a predicate describing the desired set of terms [DH94, DFH95], but do not further investigate into concrete applications. We adapt their idea to the π-calculus, and present proof-strategies for this kind of induction. In Section 3.2.2, we employ both induction-hypotheses yielded for the binders in well-formed process-abstractions (see Table 3.12 in Section 3.1.2) instantiating one of them with a fresh name. Then, in Sections 3.2.3 and 3.3.3,

we use coercion from meta-names to fresh object-names and back, in order to prevent that meta-variables be involved in the comparison of names. All the proofs applying coercion are constructive and follow the same scheme. We give a transformation-function employing two lists xs and ys: the list ys supplies fresh object-names $\mathbf{y}$ with which we replace meta-names y in the continuations of binders; xs keeps track of these instantiations, so that y can be restored whenever $\mathbf{y}$ is encountered.

Remark: All material presented in this chapter has been formalized in Isabelle/HOL; the proof-scripts are available at `http://www7.in.tum.de/~roeckl/PI/syntax.shtml`. The formal derivation of the theory of contexts is presented in a technical report [RHB00].

3.1 Formal Syntax

In the semantic description of languages with binders, *instantiation* of placeholders and *renaming* to avoid capture are essential. There are two basic ways to encode these operations, depending on the syntactic approach. In a classical *first-order* syntax, substitutions fulfill this purpose, whereas a *higher-order (abstract) syntax (HOAS)* employs β-reduction and α-conversion of an underlying λ-calculus. This distinction is independent of theorem-proving tools. When formalizing a first-order syntax one always obtains a *deep embedding*, whereas HOAS can be formalized in both deep and *shallow embeddings*. Due to the necessity of a substitution-function, the formalization and application of first-order syntax is usually tedious and prone to errors. On the other hand, first-order syntax allows the user to access free and bound variables alike, and provides him/her with a structural induction-principle. In higher-order syntax, the user does not have to define substitutions to implement instantiation and capture-avoiding renaming, but loses structural induction, because the continuations of binders are functions into rather than objects from the datatype. As an example, consider the π-calculus process $ax.\bar{b}x.0$ emitting along channel b a name received via channel a; the name x acts as a placeholder for the name that the continuation of the process delivers. In first-order syntax, it is represented by $\mathbf{ax}.\bar{\mathbf{b}}\mathbf{x}.0$, endowing all names with an equal status. In higher-order syntax, the representation is $\mathbf{a}x.\bar{\mathbf{b}}x.0$, to be read as taking a name $\mathbf{a}$ and a function $\lambda x.\, \bar{\mathbf{b}}x.0$. Here, $\mathbf{a}$ and $\mathbf{b}$ are ordinary names, whereas x is a bound variable from the underlying λ-calculus.

Names In process algebra, the set $\mathcal{N}$ of *names* is usually assumed to be countably infinite; see also Chapter 2. This is particularly important for the π-calculus, where the semantic analysis relies on an arbitrarily large amount of fresh names. The process $!(\nu b)\bar{a}b.0$, for example, continuously produces fresh names b to emit them over channel a; recall from Section 2.2 that the 'bang'-operator ! denotes *replication*. In a context such as $(\nu a)(!(\nu b)\bar{a}b.0 \,|\, !ab'.\bar{c}b'.0)$, where the received names are not necessarily discarded, a fresh name may have to be chosen for every output produced by the process. Intuitively, a name is fresh if it does not occur among the free (alternatively, free and bound) names of a process; see Section 2.2.

Like in Section 2.3.2, we use $\mathbf{a}, \mathbf{b}, \ldots$ to range over object-level names (or, object-names for short) and $a, b, \ldots$ for the meta-level names (or, meta-names). Recall from Section 2.3.2 that in a shallow embedding, meta-names and object-names cannot be confused; in its negative formulation, this means that they even cannot be compared. Yet, comparison is necessary in

P	$::=$	0	P	$::=$	0	Inaction
	$\mid$	$\tau.P$		$\mid$	$\tau.P$	Silent Prefix
	$\mid$	$\bar{\mathbf{a}}\mathbf{b}.P$		$\mid$	$\bar{\mathbf{a}}\mathbf{b}.P$	Output Prefix
	$\mid$	$\mathbf{ab}.P$		$\mid$	$\mathbf{a}x.f_P(x)$	Input Prefix
	$\mid$	$(\nu\mathbf{x})P$		$\mid$	$(\nu x)f_P(x)$	Restriction
	$\mid$	$P+Q$		$\mid$	$P+Q$	Choice
	$\mid$	$P \mid Q$		$\mid$	$P \mid Q$	Parallel Composition
	$\mid$	$[\mathbf{a}=\mathbf{b}]P$		$\mid$	$[\mathbf{a}=\mathbf{b}]P$	Matching
	$\mid$	$[\mathbf{a}\neq\mathbf{b}]P$		$\mid$	$[\mathbf{a}\neq\mathbf{b}]P$	Mismatching
	$\mid$	$!P$		$\mid$	$!P$	Replication

Table 3.1: **Deep and shallow embeddings of the π-calculus.** In a *deep* embedding, input and restriction combine names and processes, and do not involve functions, whereas in a *shallow* embedding, input and restriction are considered as binders of the underlying functional mechanism, and hence take process-abstractions as arguments.

syntactic transformations on the object-level, such as substitutions, implemented by conditionals if _ then _ else _. Applied to some meta-variable x in the scope of a binder, this yields an expression $\lambda x.\,\text{if } \mathbf{a}=x \text{ then } e_1 \text{ else } e_2$. Although $\mathbf{a}$ and x are usually meant to be distinct, the evaluation of the expression depends on which value will be inserted for x (even if it will never be instantiated). In Section 3.1.2, we discuss how syntactic transformations can be performed nevertheless, using a coercion-technique first instantiating meta-names with fresh object-names and reabstracting over them after the comparison.

In the following two sections, we introduce a deep (Section 3.1.1) and a shallow (Section 3.1.2) embedding of the π-calculus. In the shallow embedding, we replace structural induction, which the deep embedding yields naturally, by rule-induction over a well-formedness predicate.

Remark: In both sections, we reason on the level of a first-order syntax $\mathcal{P}_{fo}$ and a higher-order syntax $\mathcal{P}_{ho}$, respectively. The formalizations are straightforward implementations of the syntaxes in Isabelle/HOL, and therefore the mechanized proofs can be considered as proof-checked versions of the corresponding arguments on the level of $\mathcal{P}_{fo}$ and $\mathcal{P}_{ho}$.

3.1.1 A First-Order Syntax

In this section, we develop a straightforward deep embedding of the π-calculus. It is a direct formalization of the set of first-order processes $\mathcal{P}_{fo}$ as described by the left-hand recursive datatype in Table 3.1, which in turn is a one-to-one image of the monadic π-calculus; see the definition of $\mathcal{P}$ in Section 2.2. Every constructor in this syntax applies purely first-order arguments, even the binders input and restriction. The input operator $\mathbf{ab}.P$, for example, is of type $(\mathcal{N} \to \mathcal{N} \to \mathcal{P}_{fo}) \to \mathcal{P}_{fo}$, taking two names and a process as arguments and delivering a process. Therefore, all names occuring in a process $P \in \mathcal{P}_{fo}$ are object-variables. The definition of $\mathcal{P}_{fo}$ naturally yields structural induction, and Isabelle/HOL automatically computes the corresponding principles. A somewhat tricky part of the formalization is the implementation of a suitable notion of substitution. This general problem of deep embeddings becomes particularly

$db_{fo}(0)$	$\stackrel{\text{def}}{=}$	0	$db_{fo}(P+Q)$	$\stackrel{\text{def}}{=}$	$max(db_{fo}(P),\ db_{fo}(Q))$
$db_{fo}(\tau.P)$	$\stackrel{\text{def}}{=}$	$db_{fo}(P)$	$db_{fo}(P\,\vert\,Q)$	$\stackrel{\text{def}}{=}$	$max(db_{fo}(P),\ db_{fo}(Q))$
$db_{fo}(\bar{\mathbf{a}}\mathbf{b}.P)$	$\stackrel{\text{def}}{=}$	$db_{fo}(P)$	$db_{fo}([\mathbf{a}=\mathbf{b}]P)$	$\stackrel{\text{def}}{=}$	$db_{fo}(P)$
$db_{fo}(\mathbf{ab}.P)$	$\stackrel{\text{def}}{=}$	$1+db_{fo}(P)$	$db_{fo}([\mathbf{a}\neq\mathbf{b}]P)$	$\stackrel{\text{def}}{=}$	$db_{fo}(P)$
$db_{fo}((\nu\mathbf{b})P)$	$\stackrel{\text{def}}{=}$	$1+db_{fo}(P)$	$db_{fo}(!P)$	$\stackrel{\text{def}}{=}$	$db_{fo}(P)$

Table 3.2: **Depth of binders in first-order processes.** The function db_{fo} computes the depth of binders for first-order processes. The information is necessary for the creation of a sufficiently large supply of fresh names.

tedious for languages with a large number of operators, because there has to be an instance of the substitution-function for each of them. In this section, we use a very simplistic notion of substitution which works properly only in a limited range of application; for instance, it may deliver wrong results in an α-conversion. This suffices for our purpose, however, because we are merely interested in the normalization of processes and in proving α-equivalence.

In the sequel, we introduce the substitution-function and use it to define *normalization* of processes and *α-equivalence* upon $\mathcal{P}_{fo}$. A normalization-function replaces bound names in a process by fresh names, and hence avoids the multiple use of a name in different binders along a path in the process-tree; α-equivalence identifies processes which are equal except for their bound names, for instance a process P and its normalization P'. In Section 3.3, we show that the transformation of a first-order process P to a higher-order process and back amounts to a normalized process P', hence P and P' are α-equivalent.

Counting Binders A usual strategy for proving that two processes are α-equivalent is to compare their process-trees, instantiating the continuations of binders with fresh names, hence abstracting from the bound names of the processes. The processes $\mathbf{ab}.P$ and $\mathbf{a}'\mathbf{b}'.P'$ can be considered α-equivalent, for instance, if $\mathbf{a} = \mathbf{a}'$, and $P\{\mathbf{c}/\mathbf{b}\}$ and $P'\{\mathbf{c}/\mathbf{b}'\}$ are α-equivalent for a fresh $\mathbf{c}$. In theorem-proving, it is often convenient to produce a supply of fresh names a priori. The necessary size of such a supply depends on the number of binders along the paths in the process-tree. It can be determined by static analysis employing a recursive function db_{fo}, as defined in Table 3.2. Whenever the function encounters a binder, it is incremented by one; for instance, $db_{fo}(\mathbf{ab}.P) = 1 + db_{fo}(P)$. For branching combinators, the function naturally takes the maximum; for example, $db_{fo}(P\,|\,Q) = max(db_{fo}(P),\ db_{fo}(Q))$.

Free, bound, and fresh names As mentioned above, there is an obvious one-to-one correspondence between the monadic π-calculus and $\mathcal{P}_{fo}$. The functions computing *free* and *bound* names can therefore be directly adapted from Section 2.2.1, see Table 3.3. For instance, the free names of a process $\mathbf{ab}.P$ are computed from the free names of P by first removing $\mathbf{b}$ and consecutively adding $\mathbf{a}$, yielding $fn_{fo}(\mathbf{ab}.P) = \{\mathbf{a}\} \cup (fn_{fo}(P) - \{\mathbf{b}\})$. In our formalization in Isabelle/HOL, we use the pre-defined function `insert` to add $\mathbf{a}$ to $fn_{fo}(P) - \{\mathbf{b}\}$. The set of names of a process P is simply defined as the union of free and bound names, that is, $n_{fo}(P) = fn_{fo}(P) \cup bn_{fo}(P)$. We consider a name to be *fresh* in a process if it occurs neither among its free nor its bound names. An alternative notion of freshness might allow bound

$$
\begin{aligned}
fn_{fo}(0) &\stackrel{\text{def}}{=} \emptyset & bn_{fo}(0) &\stackrel{\text{def}}{=} \emptyset \\
fn_{fo}(\tau.P) &\stackrel{\text{def}}{=} fn_{fo}(P) & bn_{fo}(\tau.P) &\stackrel{\text{def}}{=} bn_{fo}(P) \\
fn_{fo}(\bar{\mathbf{a}}\mathbf{b}.P) &\stackrel{\text{def}}{=} \{\mathbf{a},\mathbf{b}\} \cup fn_{fo}(P) & bn_{fo}(\bar{\mathbf{a}}\mathbf{b}.P) &\stackrel{\text{def}}{=} bn_{fo}(P) \\
fn_{fo}(\mathbf{a}\mathbf{b}.P) &\stackrel{\text{def}}{=} \{\mathbf{a}\} \cup (fn_{fo}(P) - \{\mathbf{b}\}) & bn_{fo}(\mathbf{a}\mathbf{b}.P) &\stackrel{\text{def}}{=} \{\mathbf{b}\} \cup bn_{fo}(P) \\
fn_{fo}((\nu\mathbf{b})P) &\stackrel{\text{def}}{=} fn_{fo}(P) - \{\mathbf{b}\} & bn_{fo}((\nu\mathbf{b})P) &\stackrel{\text{def}}{=} \{\mathbf{b}\} \cup bn_{fo}(P) \\
fn_{fo}(P+Q) &\stackrel{\text{def}}{=} fn_{fo}(P) \cup fn_{fo}(Q) & bn_{fo}(P+Q) &\stackrel{\text{def}}{=} bn_{fo}(P) \cup bn_{fo}(Q) \\
fn_{fo}(P \mid Q) &\stackrel{\text{def}}{=} fn_{fo}(P) \cup fn_{fo}(Q) & bn_{fo}(P \mid Q) &\stackrel{\text{def}}{=} bn_{fo}(P) \cup bn_{fo}(Q) \\
fn_{fo}([\mathbf{a}=\mathbf{b}]P) &\stackrel{\text{def}}{=} \{\mathbf{a},\mathbf{b}\} \cup fn_{fo}(P) & bn_{fo}([\mathbf{a}=\mathbf{b}]P) &\stackrel{\text{def}}{=} bn_{fo}(P) \\
fn_{fo}([\mathbf{a}\neq\mathbf{b}]P) &\stackrel{\text{def}}{=} \{\mathbf{a},\mathbf{b}\} \cup fn_{fo}(P) & bn_{fo}([\mathbf{a}\neq\mathbf{b}]P) &\stackrel{\text{def}}{=} bn_{fo}(P) \\
fn_{fo}(!P) &\stackrel{\text{def}}{=} fn_{fo}(P) & bn_{fo}(!P) &\stackrel{\text{def}}{=} bn_{fo}(P)
\end{aligned}
$$

$$
n_{fo}(P) \stackrel{\text{def}}{=} fn_{fo}(P) \cup bn_{fo}(P)
$$

Table 3.3: **Free, bound, and fresh names of first-order processes.** A name is *free* in a process if it is not in the scope of an input prefix or a restriction, otherwise it is *bound*. A name is *fresh* for a process P if it is neither free nor bound in P.

names as well; yet, we are interested in α-conversion-free substitutions, hence the stricter condition.

Substitution We define only a basic notion of substitution which does not prevent name-capturing by α-conversion, but works as intended in case the substitute is fresh or equal to the substituent. It is based on a conditional rewriting names,

$$
\mathbf{a}\{\mathbf{c}/\mathbf{d}\} \stackrel{\text{def}}{=} \text{if } \mathbf{a} = \mathbf{d} \text{ then } \mathbf{c} \text{ else } \mathbf{a},
$$

and is defined primitively recursively on the level of processes. We use $P\{\mathbf{c}/\mathbf{d}\}$ for a syntax: P is a process, $\mathbf{c}$ is the substitute, and $\mathbf{d}$ is the substituent; Table 3.4 contains the whole set of defining rules. The interesting part is the definition of substitution for processes with binders, that is, $(\mathbf{ab}.P)\{\mathbf{c}/\mathbf{d}\}$ and $((\nu\mathbf{b})P)\{\mathbf{c}/\mathbf{d}\}$. Consider the defining equations for inaction, output, and input:

$$
\begin{aligned}
0\{\mathbf{c}/\mathbf{d}\} &\stackrel{\text{def}}{=} 0 \\
(\bar{\mathbf{a}}\mathbf{b}.P)\{\mathbf{c}/\mathbf{d}\} &\stackrel{\text{def}}{=} \overline{\mathbf{a}\{\mathbf{c}/\mathbf{d}\}}\,\mathbf{b}\{\mathbf{c}/\mathbf{d}\}.P\{\mathbf{c}/\mathbf{d}\} \\
(\mathbf{ab}.P)\{\mathbf{c}/\mathbf{d}\} &\stackrel{\text{def}}{=} \text{if } \mathbf{b} = \mathbf{d} \text{ then } \mathbf{a}\{\mathbf{c}/\mathbf{d}\}\mathbf{b}.P \text{ else } \mathbf{a}\{\mathbf{c}/\mathbf{d}\}\mathbf{b}.P\{\mathbf{c}/\mathbf{d}\}
\end{aligned}
$$

The definitions for inaction and output are straightforward: the one does not contain names anyway, and the other replaces the names and recursively applies the substitution to its continuation. The definition of $(\mathbf{ab}.P)\{\mathbf{c}/\mathbf{d}\}$ is more intricate. Clearly, the subject $\mathbf{a}$ is replaced as usual. For the object $\mathbf{b}$ two cases can be considered: (1) if it is equal to the substituent $\mathbf{d}$, it follows that $\mathbf{d}$ cannot occur free in P and does not have to be replaced there, hence the substitution can stop; (2) if it is different from $\mathbf{d}$, it is left unchanged, and the substitution is applied regularly to the continuation, where certainly $\mathbf{b}$ will never be replaced by $\mathbf{c}$. We formalize the above defining equations as follows:

$$
\begin{array}{rcl}
0\{\mathbf{c}/\mathbf{d}\} & \stackrel{\text{def}}{=} & 0 \\
(\tau.P)\{\mathbf{c}/\mathbf{d}\} & \stackrel{\text{def}}{=} & \tau.P\{\mathbf{c}/\mathbf{d}\} \\
(\bar{\mathbf{a}}\mathbf{b}.P)\{\mathbf{c}/\mathbf{d}\} & \stackrel{\text{def}}{=} & \overline{\mathbf{a}\{\mathbf{c}/\mathbf{d}\}}\,\mathbf{b}\{\mathbf{c}/\mathbf{d}\}.P\{\mathbf{c}/\mathbf{d}\} \\
(\mathbf{a}\mathbf{b}.P)\{\mathbf{c}/\mathbf{d}\} & \stackrel{\text{def}}{=} & \text{if } \mathbf{b}=\mathbf{d} \text{ then } \mathbf{a}\{\mathbf{c}/\mathbf{d}\}\mathbf{b}.P \text{ else } \mathbf{a}\{\mathbf{c}/\mathbf{d}\}\mathbf{b}.P\{\mathbf{c}/\mathbf{d}\} \\
((\nu\mathbf{x})P)\{\mathbf{c}/\mathbf{d}\} & \stackrel{\text{def}}{=} & \text{if } \mathbf{x}=\mathbf{d} \text{ then } (\nu\mathbf{x})P \text{ else } (\nu\mathbf{x})P\{\mathbf{c}/\mathbf{d}\} \\
(P+Q)\{\mathbf{c}/\mathbf{d}\} & \stackrel{\text{def}}{=} & P\{\mathbf{c}/\mathbf{d}\}+Q\{\mathbf{c}/\mathbf{d}\} \\
(P\,|\,Q)\{\mathbf{c}/\mathbf{d}\} & \stackrel{\text{def}}{=} & P\{\mathbf{c}/\mathbf{d}\}\,|\,Q\{\mathbf{c}/\mathbf{d}\} \\
([\mathbf{a}=\mathbf{b}]P)\{\mathbf{c}/\mathbf{d}\} & \stackrel{\text{def}}{=} & [\mathbf{a}\{\mathbf{c}/\mathbf{d}\}=\mathbf{b}\{\mathbf{c}/\mathbf{d}\}]P\{\mathbf{c}/\mathbf{d}\} \\
([\mathbf{a}\neq\mathbf{b}]P)\{\mathbf{c}/\mathbf{d}\} & \stackrel{\text{def}}{=} & [\mathbf{a}\{\mathbf{c}/\mathbf{d}\}\neq\mathbf{b}\{\mathbf{c}/\mathbf{d}\}]P\{\mathbf{c}/\mathbf{d}\} \\
(!P)\{\mathbf{c}/\mathbf{d}\} & \stackrel{\text{def}}{=} & !P\{\mathbf{c}/\mathbf{d}\}
\end{array}
$$

Table 3.4: **Substitution of names in processes.** The function $_\{_/_\}$ implements a basic notion of substitution which works correctly for *fresh* substitutes. It suffices to define α-equivalence (see Table 3.5) and normal-forms (see Table 3.7).

```
fosubst (foNull) c d = foNull
fosubst (foTau P) c d = foTau (fosubst P c d)
fosubst (foOut a b P) c d = foOut (fonsubst a c d) (fonsubst b c d)
                                  (fosubst P c d)
fosubst (foIn a b P) c d = if b=d then foIn (fonsubst a c d) b P
                                  else foIn (fonsubst a c d) b (fosubst P c d)
```

As an example for the name-capturing that can be caused by the function, consider $\mathbf{ab}.\bar{\mathbf{a}}\mathbf{c}.0$ with $\mathbf{b} \notin \{\mathbf{a}, \mathbf{c}\}$, and suppose that $\mathbf{c}$ is to be replaced by $\mathbf{b}$. Applying $_\{_/_\}$, we obtain,

$$
\begin{array}{rcll}
(\mathbf{ab}.\bar{\mathbf{a}}\mathbf{c}.0)\{\mathbf{b}/\mathbf{c}\} & = & \mathbf{ab}.(\bar{\mathbf{a}}\mathbf{c}.0)\{\mathbf{b}/\mathbf{c}\} & \text{because } \mathbf{a}\neq\mathbf{c} \text{ and } \mathbf{b}\neq\mathbf{c} \\
& = & \mathbf{ab}.\bar{\mathbf{a}}\mathbf{b}.(0)\{\mathbf{b}/\mathbf{c}\} & \text{because } \mathbf{a}\neq\mathbf{c} \\
& = & \mathbf{ab}.\bar{\mathbf{a}}\mathbf{b}.0. &
\end{array}
$$

After the transformation, the output-object is suddenly bound by the input-prefix. The resulting process is certainly not α-equivalent to the original process. Ideally, the $\mathbf{b}$ bound by the input-prefix should be converted into a fresh name $\mathbf{b}'$ before substituting $\mathbf{b}$ for $\mathbf{c}$ in the continuation. The problem is that such an α-conversion necessitates a notion of substitution, which we are currently trying to define. However, the function works well if the substitute is fresh. As a consequence, it suffices for our purpose of introducing normalization and α-equivalence. In general, it further yields a notion of capture-avoiding substitution if a normalization is performed before each regular substitution. As we have seen above, a suitable supply of fresh names can be obtained from static analysis.

We write $P\{\mathbf{c}_1/\mathbf{d}_1, \ldots, \mathbf{c}_n/\mathbf{d}_n\}$ for a sequence of substitutions $P\{\mathbf{c}_n/\mathbf{d}_n\}\ldots\{\mathbf{c}_1/\mathbf{d}_1\}$. Note that the order of the substitutions is reversed; that is, the argument $\{\mathbf{c}_n/\mathbf{d}_n\}$ is applied first, and $\{\mathbf{c}_1/\mathbf{d}_1\}$ is applied last. In our formalization, we implement $P\{\mathbf{c}_1/\mathbf{d}_1, \ldots, \mathbf{c}_n/\mathbf{d}_n\}$ in terms of a primitively recursive function `folsubst` as follows:

```
folsubst P [] = P
folsubst P (x#xs) = fosubst (folsubst P xs) (fst x) (snd x)
```

$$\frac{}{0 =_\alpha 0}\,\alpha_0 \qquad \frac{P =_\alpha P'}{\tau.P =_\alpha \tau.P'}\,\alpha_1 \qquad \frac{P =_\alpha P'}{\bar{\mathbf{a}}\mathbf{b}.P =_\alpha \bar{\mathbf{a}}\mathbf{b}'.P'}\,\alpha_2$$

$$\frac{P\{\mathbf{c}/\mathbf{b}\} =_\alpha P'\{\mathbf{c}/\mathbf{b}\}' \qquad \mathbf{c} \notin n_{fo}(P) - \{\mathbf{b}\} \qquad \mathbf{c} \notin n_{fo}(P') - \{\mathbf{b}'\}}{\mathbf{ab}.P =_\alpha \mathbf{ab}'.P'}\,\alpha_3$$

$$\frac{P\{\mathbf{c}/\mathbf{b}\} =_\alpha P'\{\mathbf{c}/\mathbf{b}\}' \qquad \mathbf{c} \notin n_{fo}(P) - \{\mathbf{b}\} \qquad \mathbf{c} \notin n_{fo}(P') - \{\mathbf{b}'\}}{(\nu\mathbf{b})P =_\alpha (\nu\mathbf{b}')P'}\,\alpha_4$$

$$\frac{P =_\alpha P' \quad Q =_\alpha Q'}{P + Q =_\alpha P' + Q'}\,\alpha_5 \qquad \frac{P =_\alpha P' \quad Q =_\alpha Q'}{P \mid Q =_\alpha P' \mid Q'}\,\alpha_6$$

$$\frac{P =_\alpha P'}{[\mathbf{a} = \mathbf{b}]P =_\alpha [\mathbf{a} = \mathbf{b}]P'}\,\alpha_7 \qquad \frac{P =_\alpha P'}{[\mathbf{a} \neq \mathbf{b}]P =_\alpha [\mathbf{a} \neq \mathbf{b}]P'}\,\alpha_8 \qquad \frac{P =_\alpha P'}{!P =_\alpha !P'}\,\alpha_1$$

Table 3.5: **α-equivalence for first-order processes.** We define α-equivalence as an inductive binary predicate on $\mathcal{P}_{fo}$.

The list `xs` in `folsubst P xs` denotes $\{\mathbf{c}_1/\mathbf{d}_1, \ldots, \mathbf{c}_n/\mathbf{d}_n\}$. Later, in an (alternative) implementation of normalization, `xs` represents a simple list of pairs to be used by a transformation-function; see below.

α-equivalence Intuitively, two processes are *α-equivalent* if they differ only in their bound names, such as $\mathbf{ab}.\bar{\mathbf{a}}\mathbf{b}.0$ and $\mathbf{ac}.\bar{\mathbf{a}}\mathbf{c}.0$. We implement α-equivalence on $\mathcal{P}_{fo}$ in terms of an inductive binary predicate $=_\alpha$; Table 3.5 shows the defining axiom and rules. Obviously, syntactic equality is strictly included. Rules α_3 and α_4 for input and restriction abstract from the bound names by comparing instantiations of the continuations with respect to a fresh name. According to this definition, clearly $\mathbf{ab}.\bar{\mathbf{a}}\mathbf{b}.0 =_\alpha \mathbf{ac}.\bar{\mathbf{a}}\mathbf{c}.0$.

As an example, consider Rule α_2: two output-processes $\bar{\mathbf{a}}\mathbf{b}.P$ and $\bar{\mathbf{a}}'\mathbf{b}'.P'$, are considered α-equivalent if $\mathbf{a} = \mathbf{a}'$ and $\mathbf{b} = \mathbf{b}'$, and also their continuations P and P' are α-equivalent. The first two conditions can be encoded implicitely:

$$\frac{P =_\alpha P'}{\bar{\mathbf{a}}\mathbf{b}.P =_\alpha \bar{\mathbf{a}}\mathbf{b}.P'}\,\alpha_2$$

Again, the interesting cases are input and restriction. In order to abstract from the bound names, the continuations are instantiated with a fresh name before comparison. To be precise, a name $\mathbf{c}$ chosen for comparing the continuations of $\mathbf{ab}.P$ and $\mathbf{ab}'.P'$ is allowed to equal $\mathbf{b}$ or $\mathbf{b}'$, depending on whether it is fresh in the other process:

$$\frac{P\{\mathbf{c}/\mathbf{b}\} =_\alpha P'\{\mathbf{c}/\mathbf{b}\}' \qquad \mathbf{c} \notin n_{fo}(P) - \{\mathbf{b}\} \qquad \mathbf{c} \notin n_{fo}(P') - \{\mathbf{b}'\}}{\mathbf{ab}.P =_\alpha \mathbf{ab}'.P'}\,\alpha_3$$

We adopt this more liberal attitude towards freshness with regard to the adequacy-proof in Section 3.3. The reason is that there we want to identify normalized processes with their originals. In the proof of α-equivalence, we use the bound name of the normal-form in order to instantiate both normalized and original process. According to the rules for input and restriction, α_3 and α_4, this is possible if the name used in the normalization is fresh in the original process. Note that this liberalization does not affect the correctness of the definition.

$$\frac{}{0 =_\alpha^{ys} 0}\ \alpha'_0 \qquad \frac{P =_\alpha^{ys} P'}{\tau.P =_\alpha^{ys} \tau.P'}\ \alpha'_1 \qquad \frac{P =_\alpha^{ys} P'}{\bar{\mathbf{a}}\mathbf{b}.P =_\alpha^{ys} \bar{\mathbf{a}}\mathbf{b}.P'}\ \alpha'_2$$

$$\frac{P\{hd(ys)/\mathbf{b}\} =_\alpha^{tl(ys)} P'\{hd(ys)/\mathbf{b}'\}}{\mathbf{ab}.P =_\alpha^{ys} \mathbf{ab}'.P'}\ \alpha'_3 \qquad \frac{P\{hd(ys)/\mathbf{b}\} =_\alpha^{tl(ys)} P'\{hd(ys)/\mathbf{b}'\}}{(\nu\mathbf{b})P =_\alpha^{ys} (\nu\mathbf{b}')P'}\ \alpha'_4$$

$$\frac{P =_\alpha^{ys} P' \quad Q =_\alpha^{ys} Q'}{P + Q =_\alpha^{ys} P' + Q'}\ \alpha'_5 \qquad \frac{P =_\alpha^{ys} P' \quad Q =_\alpha^{ys} Q'}{P \mid Q =_\alpha^{ys} P' \mid Q'}\ \alpha'_6$$

$$\frac{P =_\alpha^{ys} P'}{[\mathbf{a} = \mathbf{b}]P =_\alpha^{ys} [\mathbf{a} = \mathbf{b}]P'}\ \alpha'_7 \qquad \frac{P =_\alpha^{ys} P'}{[\mathbf{a} \neq \mathbf{b}]P =_\alpha^{ys} [\mathbf{a} \neq \mathbf{b}]P'}\ \alpha'_8 \qquad \frac{P =_\alpha^{ys} P'}{!P =_\alpha^{ys} !P'}\ \alpha_9$$

Table 3.6: **An alternative α-equivalence predicate for first-order processes.** Names from a list ys are used to instantiate bound variables. It can be shown that if ys provides a sufficient amount of distinct fresh names, $P =_\alpha^{ys} P'$ implies $P =_\alpha P'$. The easier to use predicate $=_\alpha^{ys}$ can thus be used as a proof tool for $=_\alpha$.

If either $\mathbf{b}$ or $\mathbf{b}'$ equals $\mathbf{c}$ (or maybe even both), no substitution will take place in the affected process, hence no name-capturing can occur. In case of inequality, the freshness-condition has to be fulfilled, anyway.

Mechanizing proofs of α-equivalence As we have seen, the rules α_3 and α_4 for input and restriction employ fresh names to instantiate to continuations of the two binders. These fresh names are chosen dynamically when needed. An alternative way of defining α-equivalence, which is closer to an implementation, is to compute a list of fresh names a priori by static analysis based on db_{fo} and n_{fo}. In Table 3.6, we define this implementation $=_\alpha^{ys}$ of $=_\alpha$, where ys is the list of names used to instantiate continuations of binders. The defining rules for $=_\alpha^{ys}$ are identical to those for $=_\alpha$ except for Rules α'_3 and α'_4: there the first element of ys is used without caring for freshness. As an example, the rules for output-prefix and input-prefix are (where the latter has reduced to a simpler form),

$$\frac{P =_\alpha^{ys} P'}{\bar{\mathbf{a}}\mathbf{b}.P =_\alpha^{ys} \bar{\mathbf{a}}\mathbf{b}.P'}\ \alpha'_2 \qquad \frac{P\{hd(ys)/\mathbf{b}\} =_\alpha^{tl(ys)} P'\{hd(ys)/\mathbf{b}'\}}{\mathbf{ab}.P =_\alpha^{ys} \mathbf{ab}'.P'}\ \alpha'_3$$

Of course, $=_\alpha^{ys}$ does not exactly specify $=_\alpha$: for example, $\mathbf{ac}.\bar{\mathbf{a}}\mathbf{b}.0 =_\alpha^{[\mathbf{b}]} \mathbf{ad}.\bar{\mathbf{a}}\mathbf{d}.0$, although the two processes are obviously *not* α-equivalent (indeed, they are rejected by $=_\alpha$). However, for suitable ys, we can prove that $P =_\alpha^{ys} P'$ implies $P =_\alpha P'$.

LEMMA 3.1 *Let $P, P' \in \mathcal{P}_{fo}$, and let $ys \in \mathcal{N}^n$ be a list of names such that (1) n is greater than or equal to $db_{fo}(P)$, (2) ys does not contain duplicates, (3) all names in ys are fresh to both P and P'. Then, $P =_\alpha^{ys} P'$ implies $P =_\alpha P'$.*

Proof: By rule-induction over $P =_\alpha^{ys} P'$. The proof is a straightforward case-analysis using monotonicity properties of free and bound names with respect to substitutions. In particular, we apply that $n_{fo}(P\{\mathbf{c}/\mathbf{d}\}) \subseteq \{\mathbf{c}\} \cup n_{fo}(P)$, which follows by structural induction on P. The proof has been formalized in Isabelle/HOL, and consists of about fifty lines of code. □

$$
\begin{array}{rcl@{\qquad}rcl}
[\![0]\!]^{ys}_{nm} & \stackrel{\text{def}}{=} & 0 & [\![P+Q]\!]^{ys}_{nm} & \stackrel{\text{def}}{=} & [\![P]\!]^{ys}_{nm} + [\![Q]\!]^{ys}_{nm} \\
[\![\tau.P]\!]^{ys}_{nm} & \stackrel{\text{def}}{=} & \tau.[\![P]\!]^{ys}_{nm} & [\![P \mid Q]\!]^{ys}_{nm} & \stackrel{\text{def}}{=} & [\![P]\!]^{ys}_{nm} \mid [\![Q]\!]^{ys}_{nm} \\
[\![\bar{\mathbf{a}}\mathbf{b}.P]\!]^{ys}_{nm} & \stackrel{\text{def}}{=} & \bar{\mathbf{a}}\mathbf{b}.[\![P]\!]^{ys}_{nm} & [\![[\mathbf{a}=\mathbf{b}]P]\!]^{ys}_{nm} & \stackrel{\text{def}}{=} & [\mathbf{a}=\mathbf{b}][\![P]\!]^{ys}_{nm} \\
[\![\mathbf{ab}.P]\!]^{ys}_{nm} & \stackrel{\text{def}}{=} & \mathbf{a}\,hd(ys).[\![P\{hd(ys)/\mathbf{b}\}]\!]^{tl(ys)}_{nm} & [\![[\mathbf{a}\neq\mathbf{b}]P]\!]^{ys}_{nm} & \stackrel{\text{def}}{=} & [\mathbf{a}\neq\mathbf{b}][\![P]\!]^{ys}_{nm} \\
[\![(\nu\mathbf{b})P]\!]^{ys}_{nm} & \stackrel{\text{def}}{=} & (\nu hd(ys))[\![P\{hd(ys)/\mathbf{b}\}]\!]^{tl(ys)}_{nm} & [\![!P]\!]^{ys}_{nm} & \stackrel{\text{def}}{=} & ![\![P]\!]^{ys}_{nm}
\end{array}
$$

Table 3.7: **Normalization of first-order processes.** The function $[\![_]\!]^{ys}_{nm}$ computes normalized process in which all bound names are replaced by names from ys. If ys supplies sufficiently many distinct names, the normalized and the original processes are α-equivalent.

Normalization In our adequacy proof for higher-order with respect to first-order syntax, we use a normal-form which specifies the transformation of a first-order process when translated into a higher-order process and back. In this normal-form, all names occurring in binders along a path through a process-tree are distinct. Notice that for higher-order processes, we need not—and cannot—define a notion of normalization, because there the bound names are handled entirely by the meta-level, and are therefore not accessible on the object-level. As an example, consider the process $\mathbf{ab}.\bar{\mathbf{a}}\mathbf{b}.(\nu\mathbf{b})\bar{\mathbf{a}}\mathbf{b}.0$. It has a process tree with exactly one path, along which name $\mathbf{b}$ is bound twice: once by the input-prefix $\mathbf{ab}$ in the first output-prefix $\bar{\mathbf{a}}\mathbf{b}$, and a second time by the restriction-operator $(\nu\mathbf{b})$ in the second. This process is clearly *not* in normal-form.

Our normalization-function $[\![_]\!]^{ys}_{nm}$ is parameterized over a list of names ys, such as we have encountered it in the mechanized version of α-equivalence already. Also here, we leave it to static analysis to determine a suitable ys. Table 3.7 shows the defining equations of the recursive function. Consider the defining rules for output, input, and parallel composition,

$$
\begin{array}{rcl}
[\![\bar{\mathbf{a}}\mathbf{b}.P]\!]^{ys}_{nm} & \stackrel{\text{def}}{=} & \bar{\mathbf{a}}\mathbf{b}.[\![P]\!]^{ys}_{nm} \\
[\![\mathbf{ab}.P]\!]^{ys}_{nm} & \stackrel{\text{def}}{=} & \mathbf{a}\,hd(ys).[\![P\{hd(ys)/\mathbf{b}\}]\!]^{tl(ys)}_{nm} \\
[\![P \mid Q]\!]^{ys}_{nm} & \stackrel{\text{def}}{=} & [\![P]\!]^{ys}_{nm} \mid [\![Q]\!]^{ys}_{nm}
\end{array}
$$

Output-prefix and parallel composition are treated in the usual compositional style. When encountering an input prefix, $[\![_]\!]^{ys}_{nm}$ replaces the object $\mathbf{b}$ with the first element of ys and further substitutes it for every occurrence of $\mathbf{b}$ in the continuation before continuing with the normalization. With a list $[\mathbf{c},\mathbf{d}]$ of fresh names, we obtain for our example,

$$
\begin{array}{rcll}
[\![\mathbf{ab}.\bar{\mathbf{a}}\mathbf{b}.(\nu\mathbf{b})\bar{\mathbf{a}}\mathbf{b}.0]\!]^{[\mathbf{c},\mathbf{d}]}_{nm} & = & \mathbf{ac}.[\![\bar{\mathbf{a}}\mathbf{c}.(\nu\mathbf{b})\bar{\mathbf{a}}\mathbf{b}.0]\!]^{[\mathbf{d}]}_{nm} & \text{substitute } \mathbf{c} \text{ for } \mathbf{b} \text{ in continuation} \\
 & = & \mathbf{ac}.\bar{\mathbf{a}}\mathbf{c}.[\![(\nu\mathbf{b})\bar{\mathbf{a}}\mathbf{b}.0]\!]^{[\mathbf{d}]}_{nm} & \\
 & = & \mathbf{ac}.\bar{\mathbf{a}}\mathbf{c}.(\nu\mathbf{d})[\![\bar{\mathbf{a}}\mathbf{d}.0]\!]^{[]}_{nm} & \text{substitute } \mathbf{d} \text{ for } \mathbf{b} \text{ in continuation} \\
 & = & \mathbf{ac}.\bar{\mathbf{a}}\mathbf{c}.(\nu\mathbf{d})\bar{\mathbf{a}}\mathbf{d}.[\![0]\!]^{[]}_{nm} & \\
 & = & \mathbf{ac}.\bar{\mathbf{a}}\mathbf{c}.(\nu\mathbf{d})\bar{\mathbf{a}}\mathbf{d}.0 &
\end{array}
$$

In analogy to $=^{ys}_{\alpha}$, this normalization depends on ys being a list with a sufficient amount of distinct fresh names. With this condition being fulfilled, the normalized process is α-equivalent to the original, such as in our example: $\mathbf{ab}.\bar{\mathbf{a}}\mathbf{b}.(\nu\mathbf{b})\bar{\mathbf{a}}\mathbf{b}.0 =_{\alpha} \mathbf{ac}.\bar{\mathbf{a}}\mathbf{c}.(\nu\mathbf{d})\bar{\mathbf{a}}\mathbf{d}.0$. Note that for the two binders, $\mathbf{c}$ and $\mathbf{d}$ can be chosen as ‘fresh’ names, because they do not occur in the first

process and only occur in the respective binders of the second; similarly, **b** could have been chosen, because it does not occur in the second process.

LEMMA 3.2 *Let $P \in \mathcal{P}_{fo}$ and $ys \in \mathcal{N}^n$ such that (1) $db_{fo}(P) \leq n$, (2) the names in P and ys are distinct, and (3) ys does not contain duplicates. Then, $[\![P]\!]^{ys}_{nm} =_\alpha P$.*

Proof: By induction on the structure of P. In fact, we prove a stronger result:

> *Let $P \in \mathcal{P}_{fo}$. For all $xs \in (\mathcal{N} \times \mathcal{N})^m$ and $ys \in \mathcal{N}^n$ such that $db_{fo}(P) \leq n$, the names in ys are distinct from those in P and xs, and ys does not contain duplicates, it holds that $[\![P\{xs\}]\!]^{ys}_{nm} =_\alpha P\{xs\}$.*

This strengthening is necessary, because whenever a binder is encountered, a substitution is applied according to the definitions of $[\![_]\!]^{ys}_{nm}$ (see Table 3.7) and $=_\alpha$ (see Table 3.5). As these substitutions cannot be eliminated, they have to be taken into account a priori. □

Remark: The proof of Lemma 3.2 relies on the liberal definition of Rules α_3 and α_4. Recall that they do not require absolute freshness of the substitute but allow for a reuse of the name referred to by the binder.

Note that there is a close correspondence between $[\![_]\!]^{ys}_{nm}$ and $=^{ys}_\alpha$. For our example, this means $\mathbf{ab}.\bar{\mathbf{a}}\mathbf{b}.(\nu\mathbf{b})\bar{\mathbf{a}}\mathbf{b}.0 =^{[\mathbf{c},\mathbf{d}]}_\alpha \mathbf{ac}.\bar{\mathbf{a}}\mathbf{c}.(\nu\mathbf{d})\bar{\mathbf{a}}\mathbf{d}.0$. In general, $[\![P]\!]^{ys}_{nm} =^{ys}_\alpha P$ for every ys, because of the similar derivations (see Tables 3.6 and 3.7). This can be proved in Isabelle/HOL almost automatically, applying induction and one of its automatic tactics.

Remark: The function $[\![_]\!]^{ys}_{nm}$ is not primitively recursive, because it applies a substitution to the continuations of binders before a recursive application. Therefore, the formalization in Isabelle/HOL requires a measure to guarantee monotonicity. We use the size of a process-term, adding a lemma that substitutions do not change the size of a process.

Mechanizing the derivation of normal-forms Our normalization $[\![_]\!]^{ys}_{nm}$ applies a substitution whenever it encounters a binder. This means that parts of the processes are transformed several times, by different substitutions. Here we give an alternative characterization $[\![_]\!]^{(xs,ys)}_{nm}$, which traverses the process-tree only once, in a top-down manner. Like $[\![_]\!]^{ys}_{nm}$, it uses ys as a supply of fresh names. Unlike it, however, it does not only treat bound names, but transforms every free name it encounters according to an auxiliary list xs. As a result, it simultaneously applies substitution and normalization, and yields a primitively recursive function.

A complete definition of $[\![_]\!]^{(xs,ys)}_{nm}$ is given in Table 3.8. As an example, consider the case of an input-prefix $\mathbf{ab}.P$. In the corresponding defining equation, first $\mathbf{a}$ is treated according to xs, then $\mathbf{b}$ is replaced by the first name in ys, and the pair $(hd(ys), \mathbf{b})$ is added to xs to be used in the normalization of the continuation. This results in the following rule:

$$[\![\mathbf{ab}.P]\!]^{(xs,ys)}_{nm} \stackrel{\text{def}}{=} [\![\mathbf{a}]\!]^{xs}_{nm}\, hd(ys).[\![P]\!]^{((hd(ys),\mathbf{b})xs,tl(ys))}_{nm}$$

For a name $\mathbf{a}$, the function $[\![_]\!]^{xs}_{nm}$ searches xs for the first occurrence of $\mathbf{a}$, and then substitutes the name going along with $\mathbf{a}$ in xs for $\mathbf{a}$. If $\mathbf{a}$ does not occur in xs, it is left unchanged:

$$\begin{aligned} [\![\mathbf{a}]\!]^{[]}_{nm} &\stackrel{\text{def}}{=} \mathbf{a} \\ [\![\mathbf{a}]\!]^{(\mathbf{c},\mathbf{b})xs}_{nm} &\stackrel{\text{def}}{=} \text{if } \mathbf{a} = \mathbf{b} \text{ then } \mathbf{c} \text{ else } [\![\mathbf{a}]\!]^{xs}_{nm} \end{aligned}$$

$$
\begin{array}{rcl}
\llbracket 0 \rrbracket^{(xs,ys)}_{nm} & \stackrel{\text{def}}{=} & 0 \\
\llbracket \tau.P \rrbracket^{(xs,ys)}_{nm} & \stackrel{\text{def}}{=} & \tau.\llbracket P \rrbracket^{(xs,ys)}_{nm} \\
\llbracket \bar{\mathbf{a}}\mathbf{b}.P \rrbracket^{(xs,ys)}_{nm} & \stackrel{\text{def}}{=} & \overline{\llbracket \mathbf{a} \rrbracket^{xs}_{nm}} \llbracket \mathbf{b} \rrbracket^{xs}_{nm}.\llbracket P \rrbracket^{(xs,ys)}_{nm} \\
\llbracket \mathbf{a}\mathbf{b}.P \rrbracket^{(xs,ys)}_{nm} & \stackrel{\text{def}}{=} & \llbracket \mathbf{a} \rrbracket^{xs}_{nm}\, hd(ys).\llbracket P \rrbracket^{((hd(ys),\mathbf{b})xs,tl(ys))}_{nm} \\
\llbracket (\nu\mathbf{b})P \rrbracket^{(xs,ys)}_{nm} & \stackrel{\text{def}}{=} & (\nu hd(ys))\llbracket P \rrbracket^{((hd(ys),\mathbf{b})xs,tl(ys))}_{nm} \\
\llbracket P + Q \rrbracket^{(xs,ys)}_{nm} & \stackrel{\text{def}}{=} & \llbracket P \rrbracket^{(xs,ys)}_{nm} + \llbracket Q \rrbracket^{(xs,ys)}_{nm} \\
\llbracket P \mid Q \rrbracket^{(xs,ys)}_{nm} & \stackrel{\text{def}}{=} & \llbracket P \rrbracket^{(xs,ys)}_{nm} \mid \llbracket Q \rrbracket^{(xs,ys)}_{nm} \\
\llbracket [\mathbf{a} = \mathbf{b}]P \rrbracket^{(xs,ys)}_{nm} & \stackrel{\text{def}}{=} & [\llbracket \mathbf{a} \rrbracket^{xs}_{nm} = \llbracket \mathbf{b} \rrbracket^{xs}_{nm}]\llbracket P \rrbracket^{(xs,ys)}_{nm} \\
\llbracket [\mathbf{a} \neq \mathbf{b}]P \rrbracket^{(xs,ys)}_{nm} & \stackrel{\text{def}}{=} & [\llbracket \mathbf{a} \rrbracket^{xs}_{nm} \neq \llbracket \mathbf{b} \rrbracket^{xs}_{nm}]\llbracket P \rrbracket^{(xs,ys)}_{nm} \\
\llbracket !P \rrbracket^{(xs,ys)}_{nm} & \stackrel{\text{def}}{=} & !\llbracket P \rrbracket^{(xs,ys)}_{nm}
\end{array}
$$

Table 3.8: **An alternative derivation of normal-forms.** Instead of substituting the fresh names for the bound names immediately, we add them as pairs to an auxiliary list xs, and treat every name that we encounter along the continuation according to xs.

As a consequence, every $\mathbf{b}$ in a process P which is in the scope of a binder, is replaced by the appropriate name from ys; names $\mathbf{a}$ that are free are left unchanged. It is essential to use the *first* tuple for the concerned name, because it refers to the current binder. To illustrate this, consider the normalization of the process $\mathbf{ab}.\bar{\mathbf{a}}\mathbf{b}.(\nu\mathbf{b})\bar{\mathbf{a}}\mathbf{b}.0$ with a supply of fresh names $[\mathbf{c}, \mathbf{d}]$:

$$
\begin{array}{rcl}
\llbracket \mathbf{ab}.\bar{\mathbf{a}}\mathbf{b}.(\nu\mathbf{b})\bar{\mathbf{a}}\mathbf{b}.0 \rrbracket^{([],[\mathbf{c},\mathbf{d}])}_{nm} & = & \llbracket \mathbf{a} \rrbracket^{[]}_{nm}\mathbf{c}.\llbracket \bar{\mathbf{a}}\mathbf{b}.(\nu\mathbf{b})\bar{\mathbf{a}}\mathbf{b}.0 \rrbracket^{([(\mathbf{c},\mathbf{b})],[\mathbf{d}])}_{nm} \\
& = & \mathbf{ac}.\overline{\llbracket \mathbf{a} \rrbracket^{[(\mathbf{c},\mathbf{b})]}_{nm}}\llbracket \mathbf{b} \rrbracket^{[(\mathbf{c},\mathbf{b})]}_{nm}.\llbracket (\nu\mathbf{b})\bar{\mathbf{a}}\mathbf{b}.0 \rrbracket^{([(\mathbf{c},\mathbf{b})],[\mathbf{d}])}_{nm} \\
& = & \mathbf{ac}.\bar{\mathbf{a}}\mathbf{c}.(\nu\mathbf{d})\llbracket \bar{\mathbf{a}}\mathbf{b}.0 \rrbracket^{([(\mathbf{d},\mathbf{b}),(\mathbf{c},\mathbf{b})],[])}_{nm} \\
& = & \mathbf{ac}.\bar{\mathbf{a}}\mathbf{c}.(\nu\mathbf{d})\overline{\llbracket \mathbf{a} \rrbracket^{[(\mathbf{d},\mathbf{b}),(\mathbf{c},\mathbf{b})]}_{nm}}\llbracket \mathbf{b} \rrbracket^{[(\mathbf{d},\mathbf{b}),(\mathbf{c},\mathbf{b})]}_{nm}.\llbracket 0 \rrbracket^{([(\mathbf{d},\mathbf{b}),(\mathbf{c},\mathbf{b})],[])}_{nm} \\
& = & \mathbf{ac}.\bar{\mathbf{a}}\mathbf{c}.(\nu\mathbf{d})\bar{\mathbf{a}}\mathbf{d}.0
\end{array}
$$

The resulting process is in normal-form, and indeed coincides with what one would obtain when applying $\llbracket _ \rrbracket^{ys}_{nm}$ to it. In general, we can show that for a suitable ys, the two normalizations yield syntactically equal results.

LEMMA 3.3 *Let $P \in \mathcal{P}_{fo}$ and let $ys \in \mathcal{N}^n$ be a list of names such that (1) the length of ys is greater than or equal to the depth of binders of P, (2) ys does not contain duplicates, and (3) all names in ys are fresh to P. Then, $\llbracket P \rrbracket^{ys}_{nm} = \llbracket P \rrbracket^{([],ys)}_{nm}$.*

Proof: By induction on the structure of P. In fact, we prove a stronger result:

> *For all $xs = [(\mathbf{x}_1, \mathbf{y}_1), \ldots, (\mathbf{x}_m, \mathbf{y}_m)] \in (\mathcal{N} \times \mathcal{N})^m$ and $ys \in \mathcal{N}^n$ such that (1) the length of ys is greater than or equal to the depth of binders of P, (2) $[\mathbf{y}_1, \ldots, \mathbf{y}_m]$ and ys do not contain duplicates, (3) no names in $fst(xs) = [\mathbf{x}_1, \ldots, \mathbf{x}_m]$ and ys coincide with names in $snd(xs)$, and (4) the names in $fst(xs)$ and ys are fresh to P. Then, $\llbracket P\{\mathbf{x}_1/\mathbf{y}_1, \ldots, \mathbf{x}_m/\mathbf{y}_m\} \rrbracket^{ys}_{nm} = \llbracket P \rrbracket^{(xs,ys)}_{nm}$.*

This strengthening is necessary, because every occurrence of a binder yields an additional substitution in the first normalization, and augments xs in the second. It is obvious that the

$db^{\mathbf{c}}_{ho}(0)$	$\stackrel{\text{def}}{=}$	0	$db^{\mathbf{c}}_{ho}(P+Q)$	$\stackrel{\text{def}}{=}$	$max(db^{\mathbf{c}}_{ho}(P),\ db^{\mathbf{c}}_{ho}(Q))$
$db^{\mathbf{c}}_{ho}(\tau.P)$	$\stackrel{\text{def}}{=}$	$db^{\mathbf{c}}_{ho}(P)$	$db^{\mathbf{c}}_{ho}(P \mid Q)$	$\stackrel{\text{def}}{=}$	$max(db^{\mathbf{c}}_{ho}(P),\ db^{\mathbf{c}}_{ho}(Q))$
$db^{\mathbf{c}}_{ho}(\bar{\mathbf{a}}\mathbf{b}.P)$	$\stackrel{\text{def}}{=}$	$db^{\mathbf{c}}_{ho}(P)$	$db^{\mathbf{c}}_{ho}([\mathbf{a}=\mathbf{b}]P)$	$\stackrel{\text{def}}{=}$	$db^{\mathbf{c}}_{ho}(P)$
$db^{\mathbf{c}}_{ho}(\mathbf{a}b.f_P)$	$\stackrel{\text{def}}{=}$	$1 + db^{\mathbf{c}}_{ho}(f_P b)$	$db^{\mathbf{c}}_{ho}([\mathbf{a}\neq\mathbf{b}]P)$	$\stackrel{\text{def}}{=}$	$db^{\mathbf{c}}_{ho}(P)$
$db^{\mathbf{c}}_{ho}((\nu b) f_P b)$	$\stackrel{\text{def}}{=}$	$1 + db^{\mathbf{c}}_{ho}(f_P \mathbf{c})$	$db^{\mathbf{c}}_{ho}(!P)$	$\stackrel{\text{def}}{=}$	$db^{\mathbf{c}}_{ho}(P)$

Table 3.9: **Depth of binders in higher-order processes.** The function db_{ho} computes the depth of binders for higher-order processes. The information is necessary in order to supply sufficiently many fresh names.

substitution $P\{\mathbf{x}_1/\mathbf{y}_1, \dots, \mathbf{x}_m/\mathbf{y}_m\}$ corresponds exactly to xs. In fact, in the formalization in Isabelle/HOL, there is virtually no distinction between them, because the substitution is formalized as a list of type $(\mathcal{N} \times \mathcal{N})^m$ as well.

Using the first normalization, the older occurrence of $\mathbf{b}$ is eliminated from the substitution, whereas in the second it is not. This is not so bad, however, because the second occurrence of $\mathbf{b}$ in xs does not matter anyway, and hence can be removed too without causing harm. The induction yields a tedious but straightforward case-analysis, and has been fully formalized in Isabelle/HOL. □

3.1.2 A Higher-Order Syntax

Let $\mathcal{P}_{ho}$ be the set of processes described by the datatype in the right-hand grammar of Table 3.1. The syntax contains a higher-order representation of the π-calculus. The input-operator, for example, is of type $(\mathcal{N} \to (\mathcal{N} \to \mathcal{P}_{ho})) \to \mathcal{P}_{ho}$, taking a name and a function from names to processes as arguments in order to yield a process. Therefore, both object- and meta-variables can occur in a process $P \in \mathcal{P}_{ho}$. As pointed out in Chapter 2 already, higher-order syntax faces two main problems: (1) it gives rise to exotic terms, and (2) it does not provide suitable induction-principles. In this section, we discuss how to extract those terms modelling the π-calculus, and how to obtain an adequate induction principle simultaneously. We pick the *well-formed*, or *valid*, terms from $\mathcal{P}_{ho}$ by means of an inductive predicate, in order to obtain a set $\mathcal{P}^{wf}_{ho}$ of well-formed processes. In addition, we apply a set $\mathcal{P}^{wfa}_{ho}$ of well-formed process-abstractions. The sets can equivalently be described in terms of predicates $\mathbf{Wfp}$ and $\mathbf{Wfp}_a$. In Section 3.3, we give a formalized proof that $\mathcal{P}^{wf}_{ho}$ corresponds to $\mathcal{P}_{fo}$, and hence is an adequate implementation of $\mathcal{P}$.

In the sequel, we present definitions of functions that are important in syntax analysis. Note that in contrast to first-order syntax, we do not have access to the bound names, because they are represented by meta-variables, hence cannot—and need not—define a notion of bound names.

Counting Binders The function db_{ho} computes the maximal number of binders along each path of process-trees of higher-order processes, analogously to db_{fo} for first-order processes. Again, the value is incremented whenever a binder is encountered, and the subtree with the greater number is considered for choice and parallel composition. An auxiliary name $\mathbf{c}$ is

Free names of higher-order processes:

$$
\begin{array}{rclrcl}
fn(0) & \stackrel{\text{def}}{=} & \emptyset & fn(P+Q) & \stackrel{\text{def}}{=} & fn(P) \cup fn(Q) \\
fn(\tau.P) & \stackrel{\text{def}}{=} & fn(P) & fn(P \mid Q) & \stackrel{\text{def}}{=} & fn(P) \cup fn(Q) \\
fn(\bar{\mathbf{a}}\mathbf{b}.P) & \stackrel{\text{def}}{=} & \{\mathbf{a},\mathbf{b}\} \cup fn(P) & fn([\mathbf{a}=\mathbf{b}]P) & \stackrel{\text{def}}{=} & \{\mathbf{a},\mathbf{b}\} \cup fn(P) \\
fn(\mathbf{a}x.f_P(x)) & \stackrel{\text{def}}{=} & \{\mathbf{a}\} \cup fn_a(f_P) & fn([\mathbf{a} \neq \mathbf{b}]P) & \stackrel{\text{def}}{=} & \{\mathbf{a},\mathbf{b}\} \cup fn(P) \\
fn((\nu x)f_P) & \stackrel{\text{def}}{=} & fn_a(f_P) & fn(!P) & \stackrel{\text{def}}{=} & fn(P)
\end{array}
$$

Free names of higher-order process abstractions:

$$
\begin{array}{rcl}
fn_a(f_P) & \stackrel{\text{def}}{=} & \{\, \mathbf{a} \mid \forall\, \mathbf{b}.\ \mathbf{a} \in fn(f_P(\mathbf{b}))\,\} \\
fn_{aa}(ff_P) & \stackrel{\text{def}}{=} & \{\, \mathbf{a} \mid \forall\, \mathbf{b}.\ \mathbf{a} \in fn_a(ff_P(\mathbf{b}))\,\}
\end{array}
$$

Table 3.10: **Free Names of Higher-Order Processes:** The free names of higher-order processes and process abstractions are computed by a primitively recursive function *fn* and a derived function fn_a, using each other mutually. We say that a name is free in a process abstraction, if it is free in all its instantiations.

used to instantiate the process abstractions in the continuations of input-prefix and restriction. For well-formed higher-order processes, that is, for processes corresponding to first-order processes, the result covers indeed the depth of binders. For exotic terms, it may depend on the parameter $\mathbf{c}$ what number is computed. As an example, consider the process $P \stackrel{\text{def}}{=} \mathbf{a}x.\text{if } x = \mathbf{c} \text{ then } (\nu y)\bar{\mathbf{a}}y.0 \text{ else } 0$. Then $db^{\mathbf{c}}_{ho}(P) = 2$, whereas $db^{\mathbf{d}}_{ho}(P) = 1$ for every $\mathbf{d} \neq \mathbf{c}$. This is not so bad, however, because we apply db_{ho} only in proofs about well-formed processes.

Free and Fresh Names. We compute the set of free names of higher-order processes in terms of a primitively recursive function *fn*, which is similar to the standard solution for first-order processes. The interesting part of the definition are the treatment of the process abstractions in the continuations of input-prefix and restriction:

$$
\begin{array}{rcl}
fn(\mathbf{a}x.f_P(x)) & \stackrel{\text{def}}{=} & \{\mathbf{a}\} \cup fn_a(f_P) \\
fn((\nu x)f_P) & \stackrel{\text{def}}{=} & fn_a(f_P)
\end{array}
$$

We capture these cases by means of an auxiliary function fn_a, adding a name if it occurs in all instantiations of the process-abstraction:

$$
fn_a(f_P) \stackrel{\text{def}}{=} \{\, \mathbf{a} \mid \forall\, \mathbf{b}.\ \mathbf{a} \in fn(f_P(\mathbf{b}))\,\}.
$$

For higher-order representations of π-calculus processes, that is, for terms in $\mathcal{P}^{wf}_{ho}$, the function *fn* indeed computes the same set of names as fn_{fo} does for the corresponding representation in $\mathcal{P}_{fo}$. We need not—and cannot—compute the bound names of higher-order processes, because these are part of the meta-level of the theorem-prover.

We call a name *fresh* in a process or process-abstraction, if it does not occur among its free names. This notion of freshness is different from the one we employ for first-order processes,

$$\frac{}{\mathbf{Wfp}(0)}\ \mathbf{W0} \qquad \frac{\mathbf{Wfp}(P)}{\mathbf{Wfp}(\tau.P)}\ \mathbf{W1} \qquad \frac{\mathbf{Wfp}(P)}{\mathbf{Wfp}(\bar{\mathbf{a}}\mathbf{b}.P)}\ \mathbf{W2} \qquad \frac{\mathbf{Wfp}_a(f_P)}{\mathbf{Wfp}(\mathbf{a}b.f_P(b))}\ \mathbf{W3}$$

$$\frac{\mathbf{Wfp}_a(f_P)}{\mathbf{Wfp}((\nu b)f_P(b))}\ \mathbf{W4} \qquad \frac{\mathbf{Wfp}(P) \quad \mathbf{Wfp}(Q)}{\mathbf{Wfp}(P+Q)}\ \mathbf{W5} \qquad \frac{\mathbf{Wfp}(P) \quad \mathbf{Wfp}(Q)}{\mathbf{Wfp}(P \mid Q)}\ \mathbf{W6}$$

$$\frac{\mathbf{Wfp}(P)}{\mathbf{Wfp}([\mathbf{a}=\mathbf{b}]P)}\ \mathbf{W7} \qquad \frac{\mathbf{Wfp}(P)}{\mathbf{Wfp}([\mathbf{a}\neq\mathbf{b}]P)}\ \mathbf{W8} \qquad \frac{\mathbf{Wfp}(P)}{\mathbf{Wfp}(\tau.P)}\ \mathbf{W9}$$

Table 3.11: **Well-formed higher-order processes.** The set $\mathcal{P}_{ho}^{wf}$ is defined in terms of an inductive predicate **Wfp**, which makes use of a similar predicate, $\mathbf{Wfp}_a$ for well-formed process abstractions over one argument, see Table 3.12.

where we have required that a fresh name must not occur among the bound names either.

$$\begin{aligned}
\mathbf{fresh}(\mathbf{a}, P) &\stackrel{\text{def}}{=} \mathbf{a} \notin fn(P) \\
\mathbf{fresh}_a(\mathbf{a}, f_P) &\stackrel{\text{def}}{=} \mathbf{a} \notin fn_a(f_P) \\
\mathbf{fresh}_{aa}(\mathbf{a}, ff_P) &\stackrel{\text{def}}{=} \mathbf{a} \notin fn_{aa}(ff_P)
\end{aligned}$$

The following results about free and fresh names can easily be derived.

LEMMA 3.4 *Let $P \in \mathcal{P}_{ho}$ and $f_P \in \mathcal{P}_{ho}^a$. Then:*

(a) The sets of free names $fn(P)$ and $fn_a(f_P)$ are finite.

(b) There is a name which is fresh for $fn(P)$; there is a name which is fresh for $fn_a(f_P)$.

(c) For every $n \in I\!N$, there is a set $\{\mathbf{a}_1, \ldots, \mathbf{a}_n\}$ of (distinct) names which are fresh in $fn(P)$; similarly for $fn_a(f_P)$.

Proof: (a) By induction on P employing that fn_a is defined in terms of a univarsal quantification. Item (b) is as a direct consequence of (a) by referring to the existence of countably many names. Item (c) as a corollary of (b) referring to the observation that adding an element to a finite set again yields a finite set. □

Well-Formedness As pointed out in Chapter 2, the set of higher-order processes $\mathcal{P}_{ho}$ contains exotic terms due to the strength of the logical framework offered by Isabelle/HOL. We extract the *well-formed* processes $\mathcal{P}_{ho}^{wf}$ and process-abstractions $\mathcal{P}_{ho}^{wfa}$ by means of a two-layer predicate, where **Wfp** covers the processes and is based on $\mathbf{Wfp}_a$ dealing with process-abstractions, following an idea of Despeyroux, Felty, and Hirschowitz [DFH95]. As process-abstractions employ names-abstractions, that is, functions from names to processes, we define an auxiliary predicate $\mathbf{Wfn}_a$ allowing only for constant and identity functions:

$$\frac{}{\mathbf{Wfn}_a(\lambda x.\, x)}\ \mathbf{Wn0} \qquad \frac{}{\mathbf{Wfn}_a(\lambda x.\, \mathbf{a})}\ \mathbf{Wn1}$$

We define both **Wfp** and $\mathbf{Wfp}_a$ inductively, from which Isabelle/HOL automatically computes principles for induction and case-injection. A complete formal description of the rules is given in Tables 3.11 and 3.12. Intuitively, a process in $\mathcal{P}_{ho}$ is well-formed if it is extensionally equal

$$\frac{}{\mathbf{Wfp}_a(\lambda x.\,0)}\ \mathbf{Wa0} \qquad \frac{\mathbf{Wfp}_a(f_P)}{\mathbf{Wfp}_a(\lambda x.\,\tau.f_P(x))}\ \mathbf{Wa1} \qquad \frac{\mathbf{Wfn}_a(f_\mathbf{a}) \quad \mathbf{Wfn}_a(f_\mathbf{b}) \quad \mathbf{Wfp}_a(f_P)}{\mathbf{Wfp}_a(\overline{f_\mathbf{a}(x)}f_\mathbf{b}(x).f_P(x))}\ \mathbf{Wa2}$$

$$\frac{\forall \mathbf{b}.\,\mathbf{Wfp}_a(\lambda b.\,ff_P(\mathbf{b},b)) \quad \forall \mathbf{b}.\,\mathbf{Wfp}_a(\lambda x.\,ff_P(x,\mathbf{b}))}{\mathbf{Wfp}_a(\lambda x.\,\mathbf{a}b.ff_P(x,b))}\ \mathbf{Wa3}$$

$$\frac{\forall \mathbf{b}.\,\mathbf{Wfp}_a(\lambda b.\,ff_P(\mathbf{b},b)) \quad \forall \mathbf{b}.\,\mathbf{Wfp}_a(\lambda x.\,ff_P(x,\mathbf{b}))}{\mathbf{Wfp}_a(\lambda x.\,(\nu b)ff_P(x,b))}\ \mathbf{Wa4}$$

$$\frac{\mathbf{Wfp}_a(f_P) \quad \mathbf{Wfp}_a(f_Q)}{\mathbf{Wfp}_a(\lambda x.\,f_P(x)+f_Q(x))}\ \mathbf{Wa5} \qquad \frac{\mathbf{Wfp}_a(f_P) \quad \mathbf{Wfp}_a(f_Q)}{\mathbf{Wfp}_a(\lambda x.\,f_P(x)\mid f_Q(x))}\ \mathbf{Wa6}$$

$$\frac{\mathbf{Wfn}_a(f_\mathbf{a}) \quad \mathbf{Wfn}_a(f_\mathbf{b}) \quad \mathbf{Wfp}_a(f_P)}{\mathbf{Wfp}_a(\lambda x.\,[f_\mathbf{a}(x)=f_\mathbf{b}(x)]f_P(x))}\ \mathbf{Wa7} \qquad \frac{\mathbf{Wfn}_a(f_\mathbf{a}) \quad \mathbf{Wfn}_a(f_\mathbf{b}) \quad \mathbf{Wfp}_a(f_P)}{\mathbf{Wfp}_a(\lambda x.\,[f_\mathbf{a}(x)\neq f_\mathbf{b}(x)]f_P(x))}\ \mathbf{Wa8}$$

$$\frac{\mathbf{Wfp}_a(f_P)}{\mathbf{Wfp}_a(\lambda x.\,!f_P)}\ \mathbf{Wa9}$$

Table 3.12: **Well-formed higher-order process abstractions.** The set $\mathcal{P}_{ho}^{wfa}$ is defined in terms of an inductive predicate $\mathbf{Wfp}_a$, the description of which is self-contained; that is, no further predicate is necessary for the definition.

to a higher-order representation of a process in $\mathcal{P}$. The process $\mathbf{b}x.f_W(x)$ from Section 2.3.2 is certainly well-formed, whereas $\mathbf{b}x.f_E(x)$ is not. Recall that

$$\begin{aligned} f_E &= \lambda x.\ \text{if } \mathbf{a}=x \text{ then } 0 \text{ else } \mathbf{a}b.0 \\ f_W &= \lambda x.\ \mathbf{a}b.0 \end{aligned}$$

The predicate $\mathbf{Wfp}_a$ for process-abstractions over one name is particularly interesting in that it does not rely on a further predicate for process-abstractions over two names (and so forth). Instead, its definition is self-contained, requiring for the continuations of binders only that they are well-formed in both arguments. For the input-prefix, for instance, we hence obtain: $\lambda x.\ f_\mathbf{a}(x)b.ff_P(x,b)$ is well-formed, if $f_\mathbf{a}$ is a well-formed names abstraction, and for all $\mathbf{b}$, the instantiations $\lambda x.\ ff_P(\mathbf{b},x)$ and $\lambda x.\ ff_P(x,\mathbf{b})$ are well-formed. We thus obtain two versions of the input rule for $\mathbf{Wfp}$ and $\mathbf{Wfp}_a$:

$$\frac{\mathbf{Wfp}_a(f_P)}{\mathbf{Wfp}(\mathbf{a}b.f_P(b))}\ \mathbf{W3} \qquad \frac{\forall \mathbf{b}.\,\mathbf{Wfp}_a(\lambda b.\,ff_P(\mathbf{b},b)) \quad \forall \mathbf{b}.\,\mathbf{Wfp}_a(\lambda x.\,ff_P(x,\mathbf{b}))}{\mathbf{Wfp}_a(\lambda x.\,\mathbf{a}b.ff_P(x,b))}\ \mathbf{Wa3}$$

The rules concerning the restriction-operator are defined analogously. The rules for the other operators are defined compositionally, in a standard way.

To conclude this section, we state some basic results for well-formed processes and process-abstractions. They state that every instantiation of well-formed process-abstractions yields a well-formed process, that a name that does not occur in a process-abstraction does not occur in instantiations with different names, and that it does not matter which name to use when computing the depth of binders.

LEMMA 3.5 *Let $P \in \mathcal{P}_{ho}^{wf}$ be a well-formed process, and $f_P \in \mathcal{P}_{ho}^{wfa}$ be a well-formed process abstraction. Then the following properties hold:*

(a) For every name $\mathbf{a}$, the process $f_P(\mathbf{a})$ is well-formed.

(b) *If a name* $\mathbf{a}$ *is fresh in* f_P *and* $\mathbf{a} \neq \mathbf{b}$*, then* $\mathbf{a}$ *is fresh in* $f_P(\mathbf{b})$ *as well; that is,* $\mathbf{fresh}_a(\mathbf{a}, f_P)$ *implies* $\mathbf{fresh}(\mathbf{a}, f_P(\mathbf{b}))$.

(c) *The auxiliary name for counting the binders does not matter; that is,* $db^{\mathbf{c}}_{ho}(P) = db^{\mathbf{c}'}_{ho}(P)$ *and* $db^{\mathbf{c}}_{ho}(f_P(\mathbf{b})) = db^{\mathbf{c}'}_{ho}(f_P(\mathbf{b}'))$*, for arbitrary* $\mathbf{b}, \mathbf{b}', \mathbf{c}, \mathbf{c}' \in \mathcal{N}$.

Proof: By induction on $\mathcal{P}^{wf}_{ho}$ and $\mathcal{P}^{wfa}_{ho}$, using the automatic tactics of the theorem-prover. □

In the following two sections, we use the well-formedness predicates **Wfp** and $\mathbf{Wfp}_a$ first to derive vital syntactic properties of the π-calculus (see Section 3.2), and then to prove that $\mathcal{P}^{wf}_{ho}$ is an adequate model of $\mathcal{P}$ (see Section 3.3). The proofs make extensive use of the induction-principles obtained from the definitions of **Wfp** and $\mathbf{Wfp}_a$, and rely on the extinction of exotic terms by the well-formedness predicates.

3.2 Deriving Syntactic Properties

When reasoning about the semantics of processes, a number of syntactic principles are regularly applied: terms are instantiated with values received in an input, identifiers are renamed to avoid name-capturing, or processes are compared. In shallow embeddings, instantiations and capture-avoiding renamings are generally provided by the meta-level in terms of β-reduction and α-conversion; see Section 2.3.2. In order to compare terms syntactically, Isabelle/HOL offers Leibnitz-equality on its object-level, that is, an extensional notion of α-equivalence: two functions f and g are considered to be equal, $f = g$, if they are equal for all arguments: $\forall x.\, f(x) = g(x)$. In contrast, Coq implements equality merely as the smallest reflexive relation, hence does not even consider transitivity. We will see in Section 3.2.2 that for well-formed process-abstractions Leibnitz-equality can be further specialized, so that only a single instantiation with a fresh name has to be considered. Further, in Section 3.2.3 we derive a principle for abstracting over a free name in a process. Such an abstraction necessarily involves a comparison of names; recall from above that this is generally an intricate task in higher-order syntax.

A theory of contexts In [HMS00], Honsell, Miculan, and Scagnetto present a *theory of contexts* for the π-calculus which describes three syntactic principles that are necessary for an analysis of processes in a shallow embedding but are not provided by the meta-level. Together with those principles provided by the meta-level, the three laws allow the user to derive transitions and to perform bisimulation-proofs (see [HMS00]). Let $R \in \mathcal{P}^{wf}_{ho}$ and $f_P, f_Q \in \mathcal{P}^{wfa}_{ho}$, and let $\mathbf{a}, \mathbf{b} \in \mathcal{N}$. Then,

Monotonicity: *If* $\mathbf{a}$ *is fresh in* $f_P(\mathbf{b})$ *then* $\mathbf{a}$ *is fresh in* f_P*; that is,* $\mathbf{fresh}(\mathbf{a}, f_P(\mathbf{b}))$ *implies* $\mathbf{fresh}_a(\mathbf{a}, f_P)$.

Extensionality: f_P *and* f_Q *are equal if they are equal for a fresh name; that is, if* $\mathbf{fresh}_a(\mathbf{a}, f_P)$, $\mathbf{fresh}_a(\mathbf{a}, f_Q)$*, and* $f_P(\mathbf{a}) = f_Q(\mathbf{a})$*, then* $f_P = f_Q$.

β-Expansion: *There exists a function* $f_R \in \mathcal{P}^{wfa}_{ho}$ *such that* $\mathbf{a}$ *is fresh in* f_R *and* $f_R(\mathbf{a}) = R$.

Monotonicity provides a means to determine whether a name $\mathbf{a}$ is fresh in a process-abstraction f_P, by instantiating it with an arbitrary name $\mathbf{b}$. It provides a compositional argument for the

determination of fresh names. In our formalization, it replaces the universal quantification in the definition of fn_a with the use of a single fresh name. *Extensionality of contexts* implements an extensional notion of equivalence. The law is a specialization of Leibnitz-equality over $\mathcal{P}_{ho}^{wf}$, testing only one instantiation with a fresh name instead of all possible instantiations. *β-expansion* can be regarded as the reverse of β-reduction: instead of instantiating a process-abstraction to obtain a process, one abstracts over an object-name in a process to obtain a process abstraction. This is necessary when applying transition semantics where the rules concerning restriction deal with instantiations of the continuations in the premises, such as $(\nu b) f_P(b) \xrightarrow{\mu} (\nu b) f'_P(b)$ if $f_P(\mathbf{b}) \xrightarrow{\mu} f'_P(\mathbf{b})$ for a fresh $\mathbf{b}$. As an example, a process term $\bar{\mathbf{a}}\mathbf{b}.\mathbf{b}c.\bar{\mathbf{a}}c.0$ should be transformed into $\lambda x.\, \bar{x}\mathbf{b}.\mathbf{b}c.\bar{x}c.0$. A re-instantiation with $\mathbf{a}$ again yields the original process, that is, $\bar{\mathbf{a}}\mathbf{b}.\mathbf{b}c.\bar{\mathbf{a}}c.0$.

In their formalization of the π-calculus in Coq, Honsell et al. have to—but are also able to—state the theory of contexts in terms of axioms, and justify their validity on paper. The justification is based on a structural argument exploiting that their formalization in Coq does not produce exotic terms. An alternative proof on paper, based on category-theory, is proposed by Hofmann [Hof99]. The reason for the validity of the theory of contexts in Coq as well as the impossibility to derive it within the prover is that Coq offers virtually no object-level such as Isabelle/HOL. There, the theory of contexts is only valid for well-formed processes and process-abstractions, but can also be derived within the prover. Note, however, that when extending Coq with Leibnitz-equality, the situation would become the same as in Isabelle/HOL: well-formedness predicates would have to be defined and the three principles could be derived within the tool.

Remark: Honsell et al. require that names should not be inductive, although this would be the usual intuition behind names, see Section 2.2. The reason is that inductivity yields an effective notion of equality, from which conditionals could be derived on the object-level. These conditionals could then be used to produce exotic terms, only that in their presence the theory of contexts would become inconsistent.

In Sections 3.2.1 through 3.2.3, we discuss each of the three syntactic principles and show how they can be derived for $\mathcal{P}_{ho}^{wf}$ in hypothetical proofs relying on well-formedness. Further, in Section 3.2.2 we demonstrate how exotic terms make the theory of contexts invalid.

3.2.1 Monotonicity

Consider a process-abstraction f_P that ought to be instantiated with a fresh name in the derivation of a transition, or in a bisimulation-proof. The question how to determine a fresh name for f_P is answered by the theory of contexts by instantiating the abstraction with an arbitrary name $\mathbf{b}$ and determining a fresh name for the resulting process $f_P(\mathbf{b})$. This argument is exactly dual to the determination of the free names of f_P: there, all instantiations of f_P are considered, and the intersection yields the set free names. Using only one fresh name in the computation of the free names yields an over-approximation; this implies that the resulting set includes $fn_a(f_P)$. By contraposition, a name $\mathbf{a}$ is fresh in f_P if it is fresh in $f_P(\mathbf{b})$.

Theorem 3.6 (Monotonicity) *If $\mathbf{a}$ is fresh in $f_P(\mathbf{b})$ then $\mathbf{a}$ is fresh in $f_P \in \mathcal{P}_{ho}^{wfa}$; that is, $\mathbf{fresh}(\mathbf{a}, f_P(\mathbf{b}))$ implies $\mathbf{fresh}_a(\mathbf{a}, f_P)$.*

Proof: Monotonicity is a direct consequence of the formalizations of the definitions of free and fresh names, see Section 3.1.2 and in particular Table 3.10. Isabelle/HOL proves monotonicity with one call to an automatic tactic. $\square$

3.2.2 Extensionality of Contexts

Leibnitz-equality provides an extensional notion of equivalence that allows for a syntactic comparison of process-abstractions f_P and f_Q. The theory of contexts implements a specialization according to which only a single instantiation of f_P and f_Q with a fresh name are compared. This extensionality of contexts is a crucial test for the expressiveness of an object-level, because it is inconsistent in the presence of conditionals. Consider again the process-abstractions from Section 2.3.2,

$$\begin{array}{rcl} f_E & = & \lambda x.\ \text{if } \mathbf{a} = x \text{ then } 0 \text{ else } \mathbf{a}b.0 \\ f_W & = & \lambda x.\ \mathbf{a}b.0. \end{array}$$

No matter what the definition of freshness is, any $\mathbf{b} \neq \mathbf{a}$ should be fresh for f_E and f_W. With $f_E(\mathbf{b}) = \mathbf{a}b.0 = f_W(\mathbf{b})$, one might be tempted to conclude by applying extensionality of contexts that $f_E = f_W$. On the other hand, $f_E(\mathbf{a}) = 0 \neq \mathbf{a}b.0 = f_W(\mathbf{a})$, and hence $f_E \neq f_W$.

Fortunately, such inconsistencies only arise in the presence of exotic terms like f_E, and for $\mathcal{P}_{ho}^{wfa}$ extensionality of contexts is valid. The proof makes use of the fact that there are at least uncountably many names, because it picks an auxiliary fresh name when instantiating the first of the two induction-hypotheses.

THEOREM 3.7 (EXTENSIONALITY) *$f_P \in \mathcal{P}_{ho}^{wfa}$ and $f_Q \in \mathcal{P}_{ho}^{wfa}$ are equal if they are equal for a fresh name; that is, if $\mathbf{fresh}_a(\mathbf{a}, f_P)$, $\mathbf{fresh}_a(\mathbf{a}, f_Q)$, and $f_P(\mathbf{a}) = f_Q(\mathbf{a})$, then $f_P = f_Q$.*

Proof: By rule-induction over $\mathbf{Wfp}_a$, based on the well-formedness of f_P. The interesting cases are input and restriction with a continuation $f\!f_P$, because they make use of the fact that they have two induction-hypotheses each. All other cases are routine. Together with the premises, we obtain the following goal, for both binders:

H1 **First hypothesis:** For all $\mathbf{b}$, f_Q, and $\mathbf{a}$, if f_Q is well-formed, $f\!f_P(\mathbf{b}, \mathbf{a}) = f_Q(\mathbf{a})$, and $\mathbf{a}$ is fresh in $\lambda x.\ f\!f_P(\mathbf{b}, x)$ and f_Q, then $\lambda x.\ f\!f_P(\mathbf{b}, x) = f_Q$.

H2 **Second hypothesis:** For all $\mathbf{b}$, f_Q, and $\mathbf{a}$, if f_Q is well-formed, $f\!f_P(\mathbf{a}, \mathbf{b}) = f_Q(\mathbf{a})$, and $\mathbf{a}$ is fresh in $\lambda x.\ f\!f_P(x, \mathbf{b})$ and f_Q, then $\lambda x.\ f\!f_P(x, \mathbf{b}) = f_Q$.

P **Premises:** (1) $f\!f_P$ and $f\!f_Q$ are well-formed in both arguments, (2) $\mathbf{a}$ is fresh in $f\!f_P$ and $f\!f_Q$, and (3) $\lambda x.\ f\!f_P(x, \mathbf{a}) = \lambda x.\ f\!f_Q(x, \mathbf{a})$.

C **Conclusion:** $\lambda x.\ f\!f_P(x, \mathbf{c}) = \lambda x.\ f\!f_Q(x, \mathbf{c})$ for some arbitrary $\mathbf{c}$.

We have to prove that H1, H2, and P imply C. The proof consists of two steps, applying one of the induction-hypotheses each. As a preparation, we introduce a fresh name $\mathbf{d}$; this is possible because $fn_a(f\!f_P(x, \mathbf{c}))$ and $fn_a(f\!f_Q(x, \mathbf{c}))$ are finite (Lemma 3.4(a)). The intricate part of the proof is to find suitable instantiations of the two induction-hypotheses; further, the mechanization of the justification of the freshness condition is tedious, employing a result similar to Lemma 3.5(b).

Step 1: First, we instantiate in H1 the name **b** with **d**, the process-abstraction f_Q with $\lambda x.\, \mathit{ff}_Q(\mathbf{d}, x)$, and **a** with **a**, and obtain $\lambda x.\, \mathit{ff}_P(\mathbf{d}, x) = \lambda x.\, \mathit{ff}_Q(\mathbf{d}, x)$. As a result, we show that ff_P and ff_Q are equal in an instantiation of their first argument, no matter how the second one is instantiated. We are thus free to insert the name **c** from the conclusion.

Step 2: Second, we now repeat this technique with H2, instantiating **b** with **c**, the process-abstraction f_Q with $\lambda x.\, \mathit{ff}_Q(x, \mathbf{c})$, and **a** with **d**. As a result, we obtain the conclusion $\lambda x.\, \mathit{ff}_P(x, \mathbf{c}) = \lambda x.\, \mathit{ff}_Q(x, \mathbf{c})$. □

Remark: Recall from Rules **Wa3** and **Wa4** in Table 3.12 that we considered process-abstractions ff_P as well-formed if they are well-formed for all instantiations of *both* arguments. This yields the two universally quantified induction-hypotheses from the above proof.

3.2.3 β-Expansion

The instantiation of a process-abstraction with a name is dealt with by the meta-level in terms of β-reduction. The reverse, abstracting over a name in a process, is not provided, but is necessary in the derivation of transitions involving restriction. We derive this principle of *β-expansion* in our framework. Both formulation and proof bear several pitfalls, however. First, it is essential to require explicitely that the abstraction is well-formed, otherwise the principle turns out to be too weak: to be used in combination with extensionality of contexts (Theorem 3.7), it has to respect the well-formedness conditions stated there. Second, induction cannot be applied directly, due to the existential quantifier in the conclusion ('*there exists* an abstraction'). Consider the case where $\lambda x.\, (\nu y)\, \mathit{ff}_P(y, x)$ is to be abstracted with respect to some **a** in a rule-induction over $\mathbf{Wfp}_a$. According to the premises of the resulting proof-obligations, there are $\lambda x.\, \mathit{ff}_P(\mathbf{b}, x)$ and $\lambda x.\, \mathit{ff}_P(x, \mathbf{b})$ for every **b** (see **Wa3** and **Wa4** in Table 3.12). From this one certainly cannot conclude that there exists one genuine abstraction that works for every **b**, which would be necessary to complete the argument. Finally, one has to take care never to involve meta-variables in comparisons. When trying to abstract over **a** in $\mathbf{a}b.\bar{b}\mathbf{b}.0$ in a naive way, for instance, one might go from left to right comparing every name along the path with **a**. As a result, one would obtain $\lambda x.\, xb.\overline{\text{if } b = \mathbf{a} \text{ then } x \text{ else } b}\mathbf{b}.0$. We solve this problem by first instantiating b with a fresh name **c** in $\bar{b}\mathbf{b}.0$, and then restoring it after the comparison with **a** in $\bar{\mathbf{c}}\mathbf{b}.0$. We encode this coercion-technique in a transformation-function $[\![_]\!]^{xs}_{ys}$. The list ys supplies fresh names, and xs keeps tracks of the renamings so that meta-names can be restored. Then we show that $[\![R]\!]^{xs}_{ys}$ for suitable xs and ys is well-formed, abstracts over **a** in R, and that $[\![R]\!]^{xs}_{ys}(\mathbf{a}) = R$.

A transformation-function The transformation-function $[\![_]\!]^{xs}_{ys}$ applies coercion from meta-names to object-names and back in order to prevent comparisons with meta-names: whenever a binder is encountered, the continuation is instantiated with a fresh name **y** from a list ys, but the binder (input $\mathbf{c}y.f_P(y)$ or restriction $(\nu y)\, f_P(y)$) over y is kept unchanged; instead, a substitution-obligation $(\lambda x.\, y, \mathbf{y})$ is added to a list xs, which is itself in the scope of the binder. The x in $\lambda x.\, y$ denotes the name **a** to be abstracted over, which is referred to by a names-abstraction $\lambda x.\, x$; see Table 3.13. As a result, every comparison will work on object-names only, and will yield the result 'these names differ', for the fresh **y**; as soon as this has been noticed, the meta-name is re-inserted referring to list xs. For a preparation, all free names **b** except the name **a** to be abstracted over, are added to xs as pairs $(\lambda x.\, \mathbf{b}, \mathbf{b})$, so that the

$$
\begin{array}{rcl}
[\![0]\!]^{xs}_{ys} & \stackrel{\text{def}}{=} & \lambda x.\, 0 \\
[\![\tau.P]\!]^{xs}_{ys} & \stackrel{\text{def}}{=} & \lambda x.\, \tau.[\![P]\!]^{xs}_{ys}(x) \\
[\![\bar{\mathbf{a}}\mathbf{b}.P]\!]^{xs}_{ys} & \stackrel{\text{def}}{=} & \lambda x.\, \overline{[\![\mathbf{a}]\!]^{xs}(x)}[\![\mathbf{b}]\!]^{xs}(x).[\![P]\!]^{xs}_{ys}(x) \\
[\![\mathbf{a}x.f_P(y)]\!]^{xs}_{ys} & \stackrel{\text{def}}{=} & \lambda x.\, [\![\mathbf{a}]\!]^{xs}(x)y.[\![f_P(hd(ys))]\!]^{(y,hd(ys))xs}_{tl(ys)}(x) \\
[\![(\nu x)f_P(y)]\!]^{xs}_{ys} & \stackrel{\text{def}}{=} & \lambda x.\, (\nu y)[\![f_P(hd(ys))]\!]^{(y,hd(ys))xs}_{tl(ys)}(x) \\
[\![P+Q]\!]^{xs}_{ys} & \stackrel{\text{def}}{=} & \lambda x.\, [\![P]\!]^{xs}_{ys}(x) + [\![Q]\!]^{xs}_{ys}(x) \\
[\![P \mid Q]\!]^{xs}_{ys} & \stackrel{\text{def}}{=} & \lambda x.\, [\![P]\!]^{xs}_{ys}(x) \mid [\![Q]\!]^{xs}_{ys}(x) \\
[\![[\mathbf{a}=\mathbf{b}]P]\!]^{xs}_{ys} & \stackrel{\text{def}}{=} & \lambda x.\, [[\![\mathbf{a}]\!]^{xs}(x) = [\![\mathbf{b}]\!]^{xs}(x)][\![P]\!]^{xs}_{ys}(x) \\
[\![[\mathbf{a}\neq\mathbf{b}]P]\!]^{xs}_{ys} & \stackrel{\text{def}}{=} & \lambda x.\, [[\![\mathbf{a}]\!]^{xs}(x) \neq [\![\mathbf{b}]\!]^{xs}(x)][\![P]\!]^{xs}_{ys}(x) \\
[\![!P]\!]^{xs}_{ys} & \stackrel{\text{def}}{=} & \lambda x.\, ![\![P]\!]^{xs}_{ys}(x) \\
 & & \\
[\![\mathbf{a}]\!]^{[]} & \stackrel{\text{def}}{=} & \lambda x.\, x \\
[\![\mathbf{a}]\!]^{(f_{\mathbf{a}},\mathbf{b})xs} & \stackrel{\text{def}}{=} & \text{if } \mathbf{a}=\mathbf{b} \text{ then } f_{\mathbf{a}} \text{ else } [\![\mathbf{a}]\!]^{xs}
\end{array}
$$

Table 3.13: **A transformation-function.** List xs contains information about which name has to be replaced by what, whereas list ys is a supply of fresh names to instantiate bound variables with.

function will not change them. Name **a** is left out, so that it will finally be instantiated with the meta-name x. Table 3.13 gives a formal definition of the transformation-function. While traversing the process-tree once, every name along the path is transformed according to xs; whenever a binder is encountered, the corresponding meta-name is instantiated with a fresh object-name from ys. As typical examples, consider the transformation of output and input prefix:

$$
\begin{array}{rcl}
[\![\bar{\mathbf{a}}\mathbf{b}.P]\!]^{xs}_{ys} & \stackrel{\text{def}}{=} & \lambda x.\, \overline{[\![\mathbf{a}]\!]^{xs}(x)}[\![\mathbf{b}]\!]^{xs}(x).[\![P]\!]^{xs}_{ys}(x) \\
[\![\mathbf{a}x.f_P(y)]\!]^{xs}_{ys} & \stackrel{\text{def}}{=} & \lambda x.\, [\![\mathbf{a}]\!]^{xs}(x)y.[\![f_P(hd(ys))]\!]^{(y,hd(ys))xs}_{tl(ys)}(x)
\end{array}
$$

The function traverses the process-tree once in a top-down fashion, using up ys and extending xs with each binder encountered. Like this, only object-names are compared, but the meta-names are restored after the function has passed by a name:

$$
\begin{array}{llllllllll}
\lambda x. & \ldots & (\nu b) & \mathbf{a} & b & \mathbf{a} & b & \mathbf{b} & \ldots & [\,(\lambda x.\,\mathbf{b},\mathbf{b}),\ \ldots] \\
\longrightarrow & \uparrow & & & & & & & & \\
\lambda x. & \ldots & (\nu b) & \mathbf{a} & \mathbf{c} & \mathbf{a} & \mathbf{c} & \mathbf{b} & \ldots & [\,(\lambda x.\,b,\mathbf{c}),\ (\lambda x.\,\mathbf{b},\mathbf{b}),\ \ldots] \\
\longrightarrow & & \uparrow & & & & & & & \qquad\qquad\quad\uparrow \\
\lambda x. & \ldots & (\nu b) & x & b & \mathbf{a} & \mathbf{c} & \mathbf{b} & \ldots & [\,(\lambda x.\,b,\mathbf{c}),\ (\lambda x.\,\mathbf{b},\mathbf{b}),\ \ldots] \\
\longrightarrow & & & & & & \uparrow & & & \\
\lambda x. & \ldots & (\nu b) & x & b & x & b & \mathbf{b} & \ldots & [\,(\lambda x.\,b,\mathbf{c}),\ (\lambda x.\,\mathbf{b},\mathbf{b}),\ \ldots] \\
\longrightarrow & & & & & & & \uparrow & &
\end{array}
$$

In the remainder of this section, we present a formal proof of β-expansion that we have mechanized in Isabelle/HOL. The proof consists of three parts, corresponding to the three conjectures

in the β-expansion law: well-formedness, freshness, and equality. An interesting point with regard to the equality proof is that it applies extensionality of contexts as a proof technique and therefore makes use of the preceding proofs of well-formedness and freshness.

Well-formedness The proof that the transformation of a well-formed process yields a well-formed process-abstraction consists of two steps, resulting from the two levels on which the well-formedness predicates $\mathbf{Wfp}_a$ and $\mathbf{Wfp}$ are defined: first we prove a well-formedness result for $f_P \in \mathcal{P}_{ho}^{wfa}$, which we then apply in the actual result for $P \in \mathcal{P}_{ho}^{wf}$.

In the first inference, we do not apply a transformation-list but an abstraction $\lambda x.\, f_{xs}(x)$ over such lists. The reason is that from the beginning, the transformation-list is in the scope of two binders, that is, in that of the λ-binder over the name to be abstracted over as well as in that of the process-abstraction itself. We call such an abstraction f_{xs} well-formed if it only applies well-formed names-abstractions $f\!f_{\mathbf{a}}$, that is, constant functions $\lambda xy.\, \mathbf{b}$, or identity $\lambda xy.\, x$ or $\lambda xy.\, y$:

$$\frac{}{\mathbf{Wftrl}_a(\lambda x.\, [])}\ \mathbf{Wt0} \qquad\qquad \frac{\mathbf{Wfn}_{aa}(f\!f_{\mathbf{a}}) \qquad \mathbf{Wftrl}_a(f_{xs})}{\mathbf{Wftrl}_a(\lambda x.\, (\lambda y.\, f\!f_{\mathbf{a}}(x,y),\ \mathbf{a})f_{xs}(x))}\ \mathbf{Wt1}$$

In the well-formedness result for process-abstractions, we instantiate f_{xs} either with a constant name or with the meta-variable referring to the name to be abstracted over.

LEMMA 3.8 *Let $f_P \in \mathcal{P}_{ho}^{wfa}$ and $ys \in \mathcal{N}^n$ and $\mathbf{b}, \mathbf{c} \in \mathcal{N}$, and let $f_{xs} \in \mathcal{N} \to ((\mathcal{N} \to \mathcal{N}) \times \mathcal{N})^m$ be a well-formed abstraction over transformation-lists. Then $[\![f_P(\mathbf{c})]\!]_{ys}^{f_{xs}(\mathbf{b})}$ and $\lambda x.\, ([\![f_P(\mathbf{c})]\!]_{ys}^{f_{xs}(x)}(\mathbf{b}))$ are well-formed process-abstractions.*

Proof: By rule-induction of $\mathbf{Wfp}_a$. We validate both well-formedness assumptions simultaneously by one call to an automatic tactic each. Note that neither of them could be proved independently, because the proofs for the continuations of input and restriction mutually rely on one another. □

The well-formedness result for processes requires for the transformation-list that it maps object-names to well-formed process-abstractions. This means that it should instantiate the object-names it encounters along its way through the process-tree, either with object-names or with the x from the outer λ-abstraction. Further meta-names—which occur whenever a binder is encountered and f_{xs} is augmented—are dealt with by the results of Lemma 3.10.

LEMMA 3.9 *Let $P \in \mathcal{P}_{ho}^{wf}$ and $ys \in \mathcal{N}^n$ and $xs = [(f_{\mathbf{x}_1}, \mathbf{x}_1), \dots, (f_{\mathbf{x}_m}, \mathbf{x}_m)] \in ((\mathcal{N} \to \mathcal{N}) \times \mathcal{N})^m$ such that all $f_{\mathbf{x}_i}$ are well-formed names-abstractions. Then $[\![P]\!]_{ys}^{xs}$ is a well-formed process-abstraction.*

Proof: By rule-induction of $\mathbf{Wfp}$. We validate the proof-obligations by a single call to an automatic tactic each. Only for the two binders, input and restriction, we previously refer to Lemma 3.8. □

Freshness Like well-formedness before, freshness of the transformation is derived by rule-induction over $\mathbf{Wfp}_a$ and $\mathbf{Wfp}$. Freshness states that the name to be abstracted over is indeed eliminated by the transformation. Again, we state the result for process-abstractions f_P before deriving a similar result for processes P.

LEMMA 3.10 *Let* $f_P \in \mathcal{P}_{ho}^{wfa}$ *and* $ys \in \mathcal{N}^n$ *and* $\mathbf{a}, \mathbf{b}, \mathbf{c} \in \mathcal{N}$ *with* $\mathbf{a} \neq \mathbf{b}$*, and let* $xs = [(f_{\mathbf{x}_1}, \mathbf{x}_1), \ldots, (f_{\mathbf{x}_m}, \mathbf{x}_m)] \in ((\mathcal{N} \to \mathcal{N}) \times \mathcal{N})^m$ *such that* $\mathbf{a} \neq f_{\mathbf{x}_i}(\mathbf{b})$ *for all* i*. Then* $\mathbf{a}$ *is fresh in the process* $[\![f_P(\mathbf{c})]\!]_{ys}^{xs}(\mathbf{b})$*.*

Proof: By rule-induction of $\mathbf{Wfp}_a$. We validate the single proof-obligations by calling Isabelle's automatic tactics. □

Note that in Lemma 3.10, we do not make any assumption about $\mathbf{c}$. In case $\mathbf{c} = \mathbf{a}$, it is simply abstracted over by the transformation. In the induction-proof, it denotes that name taken from the supply ys in the latest instantiation of a meta-name.

LEMMA 3.11 *Let* $P \in \mathcal{P}_{ho}^{wf}$ *and* $ys \in \mathcal{N}^n$ *and* $\mathbf{a}, \mathbf{b} \in \mathcal{N}$ *with* $\mathbf{a} \neq \mathbf{b}$*. Further, let* $xs = [(f_{\mathbf{x}_1}, \mathbf{x}_1), \ldots, (f_{\mathbf{x}_m}, \mathbf{x}_m)] \in ((\mathcal{N} \to \mathcal{N}) \times \mathcal{N})^m$ *such that* $\mathbf{a} \neq f_{\mathbf{x}_i}(\mathbf{b})$ *for all* i*. Then* $\mathbf{a}$ *is fresh in the process* $[\![P]\!]_{ys}^{xs}(\mathbf{b})$*.*

Proof: By rule-induction of $\mathbf{Wfp}$. We validate the single proof-obligations by calling Isabelle's automatic tactics. Only for the two binders, input and restriction, we previously refer to Lemma 3.10. □

Equality Equality of a re-instantiation of a suitable transformation and the original process is again proved in two steps, first by an induction on $\mathbf{Wfp}_a$ and then by another induction on $\mathbf{Wfp}$. It is the most intricate part of the proof, because it involves freshness- and well-formedness-conditions for instantiations of the continuations of binders. The reason is that in order to make the transformation-list xs representable, we apply extensionality of contexts, which we have proved in Section 3.2.2. As a consequence, we obtain a transformation-list mapping all names to themselves.

LEMMA 3.12 *Let* $f_P \in \mathcal{P}_{ho}^{wfa}$ *and* $ys \in \mathcal{N}^n$ *and* $\mathbf{a}, \mathbf{b}, \in \mathcal{N}$ *and* $xs \in ((\mathcal{N} \to \mathcal{N}) \times \mathcal{N})^m$*. If (1)* xs *is of the form* $[(\lambda x.\, \mathbf{x}_1, \mathbf{x}_1), \ldots, (\lambda x.\, \mathbf{x}_m, \mathbf{x}_m)]$ *and does not contain* $\mathbf{a}$*, (2)* $db_{ho}^{\mathbf{c}}(f_P(\mathbf{c})) \leq n$ *and* ys *consists of distinct names and does not contain* $\mathbf{a}$*, and (3)* $ys \cap \{\mathbf{x}_1, \ldots, \mathbf{x}_m\} = \emptyset$*, then* $[\![f_P(\mathbf{b})]\!]_{ys}^{xs}(\mathbf{a}) = f_P(\mathbf{b})$*.*

Proof: By rule-induction of $\mathbf{Wfp}_a$. The proof results in a tedious case-analysis, and makes use of the well-formedness and freshness results proved previously in Lemmas 3.8 and 3.10. We instantiate the continuations of binders with the first element of ys in the comparison, in order to keep the shape of xs as simple as possible. This is possible, because premise (2) guarantees freshness, and we can apply extensionality of contexts (Theorem 3.7). The Isabelle proof-script consists of about 90 lines of code. □

LEMMA 3.13 *Let* $P \in \mathcal{P}_{ho}^{wf}$ *and* $ys \in \mathcal{N}^n$ *and* $\mathbf{a} \in \mathcal{N}$ *and* $xs \in ((\mathcal{N} \to \mathcal{N}) \times \mathcal{N})^m$*. If (1)* xs *is of the form* $[(\lambda x.\, \mathbf{x}_1, \mathbf{x}_1), \ldots, (\lambda x.\, \mathbf{x}_m, \mathbf{x}_m)]$ *and does not contain* $\mathbf{a}$*, (2)* $db_{ho}^{\mathbf{c}}(P) \leq n$ *and* ys *consists of distinct names and does not contain* $\mathbf{a}$*, and (3)* $ys \cap \{\mathbf{x}_1, \ldots, \mathbf{x}_m\} = \emptyset$*, then* $[\![P]\!]_{ys}^{xs}(\mathbf{a}) = P$*.*

Proof: By rule-induction of $\mathbf{Wfp}$. Again, the proof results in a tedious case-analysis employing extensionality of contexts (Theorem 3.7) as well as Lemmas 3.8 and 3.10. The Isabelle proof-script consists of about 80 lines of code. □

$$
\begin{array}{rcl@{\qquad}rcl}
[\![0]\!]_e^{xs} & \stackrel{\text{def}}{=} & 0 & [\![0]\!]_d^{ys} & \stackrel{\text{def}}{=} & 0 \\
[\![\tau.P]\!]_e^{xs} & \stackrel{\text{def}}{=} & \tau.[\![P]\!]_e^{xs} & [\![\tau.P]\!]_d^{ys} & \stackrel{\text{def}}{=} & \tau.[\![P]\!]_d^{ys} \\
[\![\bar{\mathbf{a}}\mathbf{b}.P]\!]_e^{xs} & \stackrel{\text{def}}{=} & \overline{[\![\mathbf{a}]\!]_e^{xs}}[\![\mathbf{b}]\!]_e^{xs}.[\![P]\!]_e^{xs} & [\![\bar{\mathbf{a}}\mathbf{b}.P]\!]_d^{ys} & \stackrel{\text{def}}{=} & \bar{\mathbf{a}}\mathbf{b}.[\![P]\!]_d^{ys} \\
[\![\mathbf{a}\mathbf{b}.P]\!]_e^{xs} & \stackrel{\text{def}}{=} & [\![\mathbf{a}]\!]_e^{xs} b.[\![P]\!]_e^{[b,\mathbf{b}]xs} & [\![\mathbf{a}\mathbf{b}.P]\!]_d^{ys} & \stackrel{\text{def}}{=} & \mathbf{a}hd(ys).[\![P]\!]_d^{tl(ys)} \\
[\![(\nu\mathbf{b})P]\!]_e^{xs} & \stackrel{\text{def}}{=} & (\nu b)[\![P]\!]_e^{[b,\mathbf{b}]xs} & [\![(\nu b)P]\!]_d^{ys} & \stackrel{\text{def}}{=} & (\nu hd(ys))[\![P]\!]_d^{tl(ys)} \\
[\![P+Q]\!]_e^{xs} & \stackrel{\text{def}}{=} & [\![P]\!]_e^{xs} + [\![Q]\!]_e^{xs} & [\![P+Q]\!]_d^{ys} & \stackrel{\text{def}}{=} & [\![P]\!]_d^{ys} + [\![Q]\!]_d^{ys} \\
[\![P \mid Q]\!]_e^{xs} & \stackrel{\text{def}}{=} & [\![P]\!]_e^{xs} \mid [\![Q]\!]_e^{xs} & [\![P \mid Q]\!]_d^{ys} & \stackrel{\text{def}}{=} & [\![P]\!]_d^{ys} \mid [\![Q]\!]_d^{ys} \\
[\![[\mathbf{a}=\mathbf{b}]P]\!]_e^{xs} & \stackrel{\text{def}}{=} & [[\![\mathbf{a}]\!]_e^{xs} = [\![\mathbf{b}]\!]_e^{xs}][\![P]\!]_e^{xs} & [\![[\mathbf{a}=\mathbf{b}]P]\!]_d^{ys} & \stackrel{\text{def}}{=} & [\mathbf{a}=\mathbf{b}][\![P]\!]_d^{ys} \\
[\![[\mathbf{a}\neq\mathbf{b}]P]\!]_e^{xs} & \stackrel{\text{def}}{=} & [[\![\mathbf{a}]\!]_e^{xs} \neq [\![\mathbf{b}]\!]_e^{xs}][\![P]\!]_e^{xs} & [\![[\mathbf{a}\neq\mathbf{b}]P]\!]_d^{ys} & \stackrel{\text{def}}{=} & [\mathbf{a}\neq\mathbf{b}][\![P]\!]_d^{ys} \\
[\![!P]\!]_e^{xs} & \stackrel{\text{def}}{=} & ![\![P]\!]_e^{xs} & [\![!P]\!]_d^{ys} & \stackrel{\text{def}}{=} & ![\![P]\!]_d^{ys}
\end{array}
$$

Table 3.14: **Encoding first-order processes in higher-order syntax, and vice versa.** The functions $[\![_]\!]_e$ and $[\![_]\!]_d$ are used to mutually translate first-order and well-formed higher-order processes.

Proving β-expansion The main result follows as a corollary of Lemmas 3.9, 3.11, and 3.13. In order to abstract over a name $\mathbf{a}$ in a process $P \in \mathcal{P}_{ho}^{wf}$, we first compute ys with $db_{ho}^{\mathbf{c}}(P)$ as a list of distinct fresh names, and create $xs = [(\lambda x.\, \mathbf{b}_1, \mathbf{b}_1), \ldots, (\lambda x.\, \mathbf{b}_m, \mathbf{b}_m)]$ from the free names in P except $\mathbf{a}$, that is, from $fn(P) - \{\mathbf{a}\} = \{\mathbf{b}_1, \ldots, \mathbf{b}_m\}$. The resulting process-abstraction $[\![P]\!]_{ys}^{xs}$ is well-formed according to Lemma 3.9, does not contain $\mathbf{a}$ (Lemma 3.11), and yields P when re-instantiated (Lemma 3.13).

THEOREM 3.14 (β-EXPANSION) *For every $R \in \mathcal{P}_{ho}^{wf}$ there exists a function $f_R \in \mathcal{P}_{ho}^{wfa}$ such that $\mathbf{a}$ is fresh in f_R and $f_R(\mathbf{a}) = R$.*

Proof: Let $n = db_{ho}^{\mathbf{c}}(R)$ for an arbitrary $\mathbf{c}$, and let $fn(R) = \{\mathbf{b}_1, \ldots, \mathbf{b}_m\}$. According to Lemma 3.4(c), there exists a set $\{\mathbf{y}_1, \ldots, \mathbf{y}_n\}$ of distinct fresh names for R, from which we can build a supply-list $ys = [\mathbf{y}_1, \ldots, \mathbf{y}_n]$. Further, let xs be $[(\lambda x.\, \mathbf{b}_1, \mathbf{b}_1), \ldots, (\lambda x.\, \mathbf{b}_m, \mathbf{b}_m)]$. Then $[\![P]\!]_{ys}^{xs}$ is the desired process-abstraction: (1) it is well-formed (Lemma 3.9); (2) $\mathbf{a}$ is fresh in it (Lemma 3.11); and (3) $[\![P]\!]_{ys}^{xs}(\mathbf{a}) = R$ (Lemma 3.13). The Isabelle proof-script consists of about 20 lines, developing xs and ys, and verifying the conditions of Lemmas 3.9, 3.11, and 3.13. □

3.3 Adequacy

In this section, we present a mechanized proof that our higher-order syntax is *adequate*, that is, that the well-formed processes implement exactly the π-calculus. To our knowledge, this is the first time that an adequacy-proof for a second-order syntax has been formalized in a theorem-prover. Despeyroux, Felty, and Hirschowitz only give a justification on paper [DFH95]. In another paper, they present a mechanized adequacy-proof for a higher-order syntax where the well-formedness predicate is defined over lists of parameters [DH94]. Adequacy of two syntaxes intuitively means that for every term in one syntax there exists a corresponding term in the other, and vice versa. This can be shown in a constructive way by exhibiting functions that *encode* first-order processes into higher-order processes and *decode* them (again):

$[\![_]\!]_e : \mathcal{P}_{fo} \rightarrow \mathcal{P}_{ho}$ and $[\![_]\!]_d : \mathcal{P}_{ho} \rightarrow \mathcal{P}_{fo}$. The adequacy-proof then consists in showing that $[\![_]\!]_e$ and $[\![_]\!]_d$ are reverse. We use encoding- and decoding-functions as depicted in Table 3.14. Like in the transformation-function in the Section 3.2.3, we have to take care that in the decoding-function meta-names are not involved in comparisons. Again, we take care of this by using a transformation-list xs instead of applying a substitution whenever a binder is encountered; that is, we use the second part of the coercion-technique from the previous section is used. The first part occurs naturally in the encoding. Methodologically, the encoding of a decoding of a higher-order process corresponds exactly to the re-instantiation of a β-abstraction over an arbitrary free name. In this section, xs is defined over tuples of names instead of pairs consisting of a names-abstraction and a name, like in the proof of β-expansion in Section 3.2.3. The reason for this simpler shape of xs that in the adequacy-proof no (additional) abstraction has to be performed.

3.3.1 The Encoding-Function

We use a function $[\![_]\!]_e$ in order to translate first-order processes from $\mathcal{P}_{fo}$ into their higher-order counterparts in $\mathcal{P}_{ho}$, or rather in $\mathcal{P}_{ho}^{wf}$; see Table 3.14 for a complete definition. The encoding follows our usual scheme: an auxiliary list xs tells for a process P in $[\![P]\!]_e^{xs}$ how its free names should be transformed. Whenever the function encounters an input or a restriction, the corresponding bound name is added to xs together with a new meta-variable, and is bound by Isabelle's functional mechanism:

$$[\![\mathbf{ab}.P]\!]_e^{xs} \stackrel{\text{def}}{=} [\![\mathbf{a}]\!]_e^{xs} b.[\![P]\!]_e^{[b,\mathbf{b}]xs} \qquad\qquad [\![(\nu\mathbf{b})P]\!]_e^{xs} \stackrel{\text{def}}{=} (\nu b)[\![P]\!]_e^{[b,\mathbf{b}]xs}$$

The list xs therefore fulfills the same functionality as xs in the transformation-function for β-expansion in Section 3.2.3. The encoding of names, $[\![a]\!]_e^{xs}$, follows our usual scheme: if $\mathbf{a}$ does not occur in xs, it is left unchanged; otherwise, it is mapped to the name accompanying its first occurrence:

$$\begin{aligned}
[\![\mathbf{a}]\!]_e^{[]} &\stackrel{\text{def}}{=} \mathbf{a} \\
[\![\mathbf{a}]\!]_e^{(\mathbf{c},\mathbf{b})xs} &\stackrel{\text{def}}{=} \text{if } \mathbf{a} = \mathbf{b} \text{ then } \mathbf{c} \text{ else } [\![\mathbf{a}]\!]_e^{xs}
\end{aligned}$$

This emphasis on the *first* occurrence is necessary again, because in non-normalized processes, a name $\mathbf{b}$ can occur under several binders. To illustrate this, consider the process $\mathbf{ab}.\bar{\mathbf{a}}\mathbf{b}.(\nu\mathbf{b})\bar{\mathbf{a}}\mathbf{b}.0$. In it, the $\mathbf{b}$ from the first output is bound by the input, whereas that in the second output is bound by the restriction. This yields the following encoding:

$$\begin{aligned}
[\![\mathbf{ab}.\bar{\mathbf{a}}\mathbf{b}.(\nu\mathbf{b})\bar{\mathbf{a}}\mathbf{b}.0]\!]_e^{[]} &= [\![\mathbf{a}]\!]_e^{[]} b.\underline{[\![\bar{\mathbf{a}}\mathbf{b}.(}\nu\mathbf{b})\bar{\mathbf{a}}\mathbf{b}.0]\!]_e^{[(b,\mathbf{b})]} && \text{new meta-name } b \\
&= \mathbf{a}b.\overline{[\![\mathbf{a}]\!]_e^{[(b,\mathbf{b})]}}[\![\mathbf{b}]\!]_e^{[b,\mathbf{b}]}.[\![(\nu\mathbf{b})\bar{\mathbf{a}}\mathbf{b}.0]\!]_e^{[b,\mathbf{b}]} \\
&= \mathbf{a}b.\bar{\mathbf{a}}b.(\nu b')\underline{[\![\bar{\mathbf{a}}\mathbf{b}.0]\!]_e^{[(b',\mathbf{b}),\,(b,\mathbf{b})]}} && \text{new meta-name } b' \\
&= \mathbf{a}b.\bar{\mathbf{a}}b.(\nu b')\overline{[\![\mathbf{a}]\!]_e^{[(b',\mathbf{b}),\,(b,\mathbf{b})]}}[\![\mathbf{b}]\!]_e^{[(b',\mathbf{b}),\,(b,\mathbf{b})]}.[\![0]\!]_e^{[(b',\mathbf{b}),\,(b,\mathbf{b})]} \\
&= \mathbf{a}b.\bar{\mathbf{a}}b.(\nu b')\bar{\mathbf{a}}b'.0
\end{aligned}$$

Recall from Section 2.3.2 that the meta-names b and b' are chosen by the meta-level and cannot be manipulated on the object-level. For instance, we cannot tell whether they are equal to or distinct from some object-name or some other meta-name.

Well-Formedness We have defined $[\![_]\!]_e$ as a function of type $\mathcal{P}_{fo} \to \mathcal{P}_{ho}$, but have already indicated that in fact it has a domain $\mathcal{P}_{ho}^{wf}$. Recall from Section 3.1.2 that we consider $\mathcal{P}_{ho}^{wf}$ as the set of processes in $\mathcal{P}_{ho}$ that are well-formed. We therefore have to prove that for every $P \in \mathcal{P}_{fo}$, the encoding $[\![P]\!]_e^{[]}$ is well-formed. In order to capture the growth of the transformation-list, we prove a more general form of the result, considering $[\![P]\!]_e^{xs}$ for an arbitrary $xs \in (\mathcal{N} \times \mathcal{N})^m$. With xs replacing object-names with meta-names, we have to reason with functions $f_{xs} \in \mathcal{N} \to (\mathcal{N} \times \mathcal{N})^m$. Again, these functions have to satisfy a well-formedness criterion, which is defined in terms of an inductive predicate:

$$\frac{}{\mathbf{Wfnml}_a(\lambda x.\,[])}\ \mathbf{Wfnml1} \qquad\qquad \frac{\mathbf{Wfn}_a(f_{\mathbf{a}}) \quad \mathbf{Wfnml}_a(xs)}{\mathbf{Wfnml}_a(\lambda x.\,(f_{\mathbf{a}}(x), \mathbf{a})xs)}\ \mathbf{Wfnml2}$$

Note that we only have to abstract over the one meta-name; all other names in f_{xs} are instantiations. The reason is that we reason about one binder at a time.

LEMMA 3.15 *For every $P \in \mathcal{P}_{fo}$ and every $xs \in (\mathcal{N} \times \mathcal{N})^m$, the encoding $[\![P]\!]_e^{xs}$ is well-formed. In particular, $[\![P]\!]_e$ is well-formed.*

Proof: By induction on the structure of P. In fact, we show that every first-order process can be transformed into a well-formed process-abstraction. By structural induction on P, we derive that for an arbitrary list-abstraction f_{xs} with $\mathbf{Wfnml}_a(f_{xs})$, the encoding $\lambda x.\,[\![P]\!]_e^{f_{xs}(x)}$ is a well-formed process-abstraction. The script of this proof consists of a few lines only, using Isabelle's automatic tactics. □

3.3.2 The Decoding-Function

We use a function $[\![_]\!]_d$ in order to translate higher-order processes from $\mathcal{P}_{ho}$ into first-order counterparts in $\mathcal{P}_{fo}$; see Table 3.14 for a complete definition. As we will see in Section 3.3.3, processes in $\mathcal{P}_{ho}^{wf}$ are translated without structural changes, so that they can be completely restored by an application of $[\![_]\!]_e$. Like the normalization-functions from Section 3.1.1, the decoding applies a list ys supplying object-names to instantiate the meta-names in the scope of input-prefixes and restrictions. For the higher-order processes, this instantiation is a simple function-application carried out by the meta-level. Object-names as they occur in output-prefixes, for instance, are left unchanged. Consider the resulting higher-order process from our previous example, $\mathbf{a}b.\bar{\mathbf{a}}b.(\nu b')\bar{\mathbf{a}}b'.0$, and a list $[\mathbf{b}, \mathbf{b}']$ of object-names. Then we obtain a first-order decoding,

$$\begin{array}{rcll}
[\![\mathbf{a}b.\bar{\mathbf{a}}b.(\nu b')\bar{\mathbf{a}}b'.0]\!]_d^{[\mathbf{b},\mathbf{b}']} & = & \mathbf{ab}.[\![\bar{\mathbf{a}}\mathbf{b}.(\nu b')\bar{\mathbf{a}}b'.0]\!]_d^{[\mathbf{b}']} & \text{instantiate } b \text{ with } \mathbf{b} \\
 & = & \mathbf{ab}.\bar{\mathbf{a}}\mathbf{b}.[\![(\nu b')\bar{\mathbf{a}}b'.0]\!]_d^{[\mathbf{b}']} & \\
 & = & \mathbf{ab}.\bar{\mathbf{a}}\mathbf{b}.(\nu \mathbf{b}')[\![\bar{\mathbf{a}}\mathbf{b}'.0]\!]_d^{[]} & \text{instantiate } b' \text{ with } \mathbf{b}' \\
 & = & \mathbf{ab}.\bar{\mathbf{a}}\mathbf{b}.(\nu \mathbf{b}')\bar{\mathbf{a}}\mathbf{b}'.0. &
\end{array}$$

The resulting process is α-equivalent to the process $\mathbf{ab}.\bar{\mathbf{a}}\mathbf{b}.(\nu\mathbf{b})\bar{\mathbf{a}}\mathbf{b}.0$ fed into the encoding-function in Section 3.3.1; see Table 3.5 for a definition of $=_\alpha$.

3.3.3 Proving Adequacy

The adequacy-theorem consists of two parts, one translating first-order into higher-order processes and back, and the other translating well-formed higher-order into first-order processes

and back. As we have illustrated by an example in the previous section, this transformation can be safely applied to well-formed processes. Exotic terms, on the other hand, can lose structural information through an instantiation. Consider $\mathbf{b}x.f_E$ from Section 2.3.2 with,

$$f_E \quad = \quad \lambda x.\ \text{if } \mathbf{a} = x \text{ then } 0 \text{ else } \mathbf{a}b.0.$$

A transformation with $ys = [\mathbf{a}, \mathbf{y}]$ yields $\mathbf{ba}.0$, and another transformation with $ys = [\mathbf{x}, \mathbf{y}]$ for a fresh $\mathbf{x}$ yields $\mathbf{bx}.\mathbf{ay}.0$. Obviously, the conditional cannot be recovered by $[\![_]\!]_e$.

THEOREM 3.16 (ADEQUACY) *The sets of processes $\mathcal{P}_{fo}$ and $\mathcal{P}_{ho}$ correspond as follows:*

1. *Let $P \in \mathcal{P}_{fo}$ and $ys \in \mathcal{N}^n$ such that $db_{fo}(P) \leq n$, the names in P and ys are distinct, and ys does not contain duplicates. Then, $[\![[\![P]\!]_e^{[]}]\!]_d^{ys} =_\alpha P$.*

2. *Let $P \in \mathcal{P}_{ho}$ be well-formed, and $ys \in \mathcal{N}^n$ such that $db^{\mathbf{c}}_{ho}(P) \leq n$, the free names in P and ys are distinct, and ys does not contain duplicates. Then, $[\![[\![P]\!]_d^{ys}]\!]_e^{[]} = P$.*

Proof: *1.* We prove by induction on the structure of P that for arbitrary xs and ys we have $[\![[\![P]\!]_e^{xs}]\!]_d^{ys} = [\![P]\!]_{nm}^{(xs,ys)}$. Then, we use the result from Lemma 3.3 to conclude that for a suitable ys, as assumed in the premises, $[\![[\![P]\!]_e^{[]}]\!]_d^{ys} = [\![P]\!]_{nm}^{ys}$. Referring to Lemma 3.2, we can infer $[\![[\![P]\!]_e^{[]}]\!]_d^{ys} =_\alpha P$.

2. The proof consists of two parts and proceeds by induction on the well-formedness predicates for process-abstractions and processes. The proof is a tedious case-analysis. It makes use of extensionality of contexts from Section 3.2.2, so that additional lemmas about the freshness of names are necessary, which are proved by induction on well-formed process-abstractions. Again, the application of a transformation-list necessitates the proofs of stronger results:

(a) *For $f_P \in \mathcal{P}_{ho}^{wfa}$ and $\mathbf{b} \in \mathcal{N}$ and $ys \in \mathcal{N}^n$ and $xs \in (\mathcal{N} \times \mathcal{N})^m$, the following holds: if (1) $\mathbf{b}$ is fresh for f_P and does not occur in ys, and (2) ys has no duplicates and $db^{\mathbf{c}}_{ho}(f_P(\mathbf{c})) \leq n$, (3) the names in ys are fresh for f_P, and (4) $xs = [(\mathbf{x}_1, \mathbf{x}_1), \ldots, (\mathbf{x}_m, \mathbf{x}_m)]$ for some $\mathbf{x}_i$, then $\lambda x.\, [\![[\![f_P(\mathbf{b})]\!]_d^{ys}]\!]_e^{(x,\mathbf{b})xs} = f_P$.*

(b) *For $P \in \mathcal{P}_{ho}^{wf}$ and $ys \in \mathcal{N}^n$ and $xs \in (\mathcal{N} \times \mathcal{N})^m$, the following holds: if (1) ys has no duplicates and $db^{\mathbf{c}}_{ho}(f_P(\mathbf{c})) \leq n$, (2) the names in ys are fresh for P, and (3) $xs = [(\mathbf{x}_1, \mathbf{x}_1), \ldots, (\mathbf{x}_m, \mathbf{x}_m)]$ for some $\mathbf{x}_i$, then $\lambda x.\, [\![[\![P]\!]_d^{ys}]\!]_e^{xs} = f_P$.*

We can assume that $xs = [(\mathbf{x}_1, \mathbf{x}_1), \ldots, (\mathbf{x}_m, \mathbf{x}_m)]$ by referring to extensionality of contexts in the proof of part *(a)*. There, in order to prove $\lambda x.\, [\![[\![f_P(\mathbf{b})]\!]_d^{ys}]\!]_e^{(x,\mathbf{b})xs} = f_P$, we exploit the freshness of $\mathbf{b}$, and instantiate both sides with it, obtaining $[\![[\![f_P(\mathbf{b})]\!]_d^{ys}]\!]_e^{(\mathbf{b},\mathbf{b})xs} = f_P(\mathbf{b})$. Applying extensionality of contexts, the proof reduces to freshness and well-formedness arguments.

We have mechanized the whole adequacy-proof in Isabelle/HOL. $\square$

3.4 Discussion

In this section, we have presented a fully formalized theory of higher-order abstract syntax for the π-calculus in Isabelle/HOL: we have introduced a straightforward deep embedding as

well as a shallow embedding; for the shallow embedding, we have introduced well-formedness predicates to obtain induction and exclude exotic terms; we have used the formalization to mechanically derive the theory of contexts; finally, we have presented an adequacy-proof relating the deep and the shallow embedding. In the remainder of this chapter, we discuss some related issues, hint at related work, and point out directions for future work.

First-order and higher-order syntax Binders generally play a crucial role in programming-languages and process-calculi. There are two basic approaches for reasoning about them on a formal logical level: in first-order syntax, free and bound parameters are equally treated as first-class objects, whereas in higher-order (abstract) syntax, bound parameters are handled by an underlying functional mechanism. Both approaches have their advantages and drawbacks. While first-order syntax naturally yields principles for structural induction, it necessitates substitutions to perform instantiations and avoid name-capturing. Both definition and application of suitable notions of substitution are tedious and error-prone, so that efficient reasoning about semantics is hardly possible. In higher-order syntax, instantiations and the treatment of bound variables are for free, yet structural induction fails, and exotic terms can arise. Therefore, syntax-analysis cannot be performed directly.

First-order syntax always yields a deep embedding in a theorem-prover. When using higher-order syntax, the user can choose between two alternative approaches, see also Section 2.3.2: either he/she formalizes a λ-calculus in a deep embedding and uses it as an underlying functional framework, or he/she applies a shallow embedding. We have followed the second approach.

Well-formedness predicates A way out of the dilemmas of HOAS is to replace the missing structural induction-principles by rule-induction based on a predicate subsuming the set of terms one is interested in. As a result, exotic terms are excluded simultaneously. A straightforward solution is to introduce a predicate over $\lambda x_1, \ldots, x_n. t$ for each n, as proposed in [DH94]. This is hard to apply in theorem-proving, however, because $x_1, \ldots, x_n$ have to be formalized in terms of finite lists on the object-level. Like related datatypes, lists are often tedious to handle in concrete proofs, see also Chapter 5. A more convenient solution from a theorem-proving point of view is therefore the application of a two-step predicate, as proposed in [DFH95]. In this chapter, we have adapted this approach for the π-calculus—yielding the set $\mathcal{P}_{ho}^{wf}$—and have fully investigated it within Isabelle/HOL.

The use of well-formedness predicates yields hypothetical judgements and proofs, because certain properties are inconsistent for exotic terms (such as extensionality of contexts in Section 3.2.2) or have to be strengthened for well-formed terms (such as β-expansion in Section 3.2.3).

Deep embeddings of the π-calculus There is a range of deep embeddings of the π-calculus, following different paradigms. Our straightforward approach from Section 3.1.1 has been adopted in [Mel95, AM96b, Mam99]. In order to prevent capture-avoiding renamings in instantiations, the McKinna-Pollack methodology [MP99] considers free and bound names as different types; it is used in [Hen99]. DeBruijn indices [deB72] are used in [Hir97]; they provide a canonical basis for bound names in a term by replacing them with the depth of the binder they are associated with. In all of these formalizations, substitution-functions have to be formalized to implement instantiations and probably capture-avoiding renamings. This makes them useful in syntax-analysis but hard to deal with in semantic arguments.

Deep embeddings of higher-order syntaxes of the π-calculus are applied in [GM96, Gay00]. While the former focusses on the approach itself and uses the π-calculus as an exemplaric application, the latter employs it as a basis for a study of type-systems. It will have to be investigated how the approach applies in a semantic analysis of the π-calculus, considering transition-systems and bisimulations.

Shallow embeddings of the π-calculus To our knowledge, three shallow embeddings of the π-calculus have been studied so far [Mil92a, HMS00, Des00]. While the first two of them rely on the meta-levels of λProlog [NM98] and Coq [BBC+99] to exclude exotic terms automatically, the last one does not consider the issue of well-formedness. As pointed out in [HMS00, Hof99], the absence of exotic terms is essential for a semantic analysis of π-calculus processes in terms of labelled transition-systems and bismulations, because it regularly applies extensionality of contexts (see [HMS00]).

The theory of contexts is necessary, because derivatives of input-transitions are instantiated with (probably fresh) names, and occasionally re-abstracted over in bisimulation proofs. In [Des00], a semantics based on abstractions and concretions [Mil91] is used. It will have to be studied which role fresh names play in this framework, and whether the theory of contexts is necessary in the analysis of this kind of semantics or not (we consider that something similar will be necessary at least).

Semantic analysis of the π-calculus in a shallow embedding The formalization presented in this chapter provides a platform for the analysis of semantics for the π-calculus in a shallow embedding in Isabelle/HOL. The framework gives the user access to the well-formedness conditions ruling out exotic terms, because the underlying predicates are implemented on the object-level. In specific logical frameworks like λProlog [NM98], or Elf [Pfe89] and its successor Twelf [PS99], the user cannot access these conditions, because they are part of the meta-level. On the other hand, our formalization requires the user to work with hypothetical judgements, whereas λProlog and Elf/Twelf relieve him/her from this task. It is an issue of future work which of the approaches is most suitable for the semantic analysis of the π-calculus and for proofs about concrete systems modelled in it.

Chapter 4

Encoding Algol in the π-calculus

The π-calculus is often referred to as the essence of concurrent languages, such as the λ-calculus is for sequential ones. Previous work gives evidence that the π-calculus can model references, functions, objects, and various forms of (non-atomic) parallelism [Wal95, Jon93, KS98, Mil92b]. Yet, so far only limited forms of combinations have been considered, and it is far from obvious that the respective π-calculus descriptions can be combined in a faithful model. *Concurrent Idealized Algol* (CIA) extends classical *Idealized Algol* (IA) with concurrency [Rey81, Bro96], and therefore integrates in an elegant way imperative, concurrent, and higher-order features. IA has been extensively studied in the literature, serving as a benchmark to various semantic models [OT97]. It has turned out that operationally-based models [AM96a, AM99, Pit96] best capture the semantics of Algol, because they do not produce *snapback-effects*. Models representing states by functions—usually denotational models do so—suffer from these effects, in that state-changes are reversed; this necessitates an application of logical relations to remove the snapback-parts from the model [OT97].

In this chapter, we investigate a π-calculus model of CIA. Our work belongs to a strand of research on giving operational semantics to IA [AM96a, AM99, Pit96], and is closely related to Abramsky and McCusker's game semantics for IA. Our main reasons for using the π-calculus are the following: (1) it has a first-order syntax, and (therefore) offers reasonably applicable behavioural equivalences; (2) it nevertheless captures higher-order semantics; (3) it offers a well-developed proof machinery including algebraic reasoning and type-systems; and, (4) by representing states as processes, it does not produce snapback-effects. The point mentioned in item (1) is a prevailing issue for languages with a higher-order syntax, as pointed out by Ferreira, Hennessy, and Jeffery [FHJ95]. Proofs of process equivalences are complicated by universal quantifications over terms. Further, it is generally hard to establish that a notion of bisimilarity for higher-order languages is a congruence: this is usually proved using Howe's technique [How96]; yet, attempts to extend this methodology to languages with local state—like CIA, in which local variables can be defined—have been unsuccessful so far.

Remark: There are decisive similarities between local names in the π-calculus and local references in imperative languages. This becomes evident in the denotational semantics of these languages: the mathematical techniques employed in modelling the π-calculus [Sta96, FMS96] were originally developed for the semantic description of local references. Yet, names and references behave differently: receiving from a channel is destructive—it consumes a value—whereas reading from a reference is not; a reference has a unique location, whereas a channel may be used by several processes for both reading and writing; et cetera. Hence, it is unclear

if and how interesting properties of imperative languages can be proved via a translation into the π-calculus.

In this chapter, we develop a π-calculus semantics for CIA in two stages: we first consider an extension of IA with parallel composition of commands (CIA-{**await**}), before adding an **await**-operator implementing atomicity (full CIA). Section 4.1 introduces a straightforward operational semantics for CIA implementing a behavioural congruence based on bisimulations. In Section 4.2, we present the encoding and show via an operational correspondence that it is sound. Section 4.3 discusses proofs of several laws and examples, including the validation of a buffer implemented by a higher-order procedure (Section 4.3.1).

Remark: Material from this chapter is presented in [RS99, Röc99].

4.1 Concurrent Idealized Algol

Idealized Algol (IA) was introduced by Reynolds as the core of Algol 60, extending a simply-typed call-by-name λ-calculus with a concept of state, implemented by means of local and global variables [Rey81]. Later, Concurrent Idealized Algol (CIA) was created by Brookes by further adding parallelism [Bro96]. One of the strengths of IA as well as CIA is the clear separation between phrases of *passive* and *active* types: while expressions can only read from variables, and are therefore passive, commands are capable of writing to them, and are therefore active. The fact that expressions are passive can make the semantic description harder, as pointed out by Abramsky and McCusker [AM96a, AM99].

Syntax We follow the syntax, typing, and notational conventions as applied in [Rey81, Bro96]. *Data-types* in IA and CIA consist of integers and booleans; *phrase-types* are constructible from variables, expressions, and commands using arrow-type:

$$\tau ::= \mathit{int} \mid \mathit{bool} \qquad \text{Data Types}$$
$$\sigma ::= \mathbf{var}[\tau] \mid \mathbf{exp}[\tau] \mid \mathbf{comm} \mid (\sigma \to \sigma) \qquad \text{Phrase Types}$$

We consider phrase-types of arbitrarily higher order, as denoted by the arrow-type $\sigma \to \sigma$ in their definition. For simplicity we omit tupling. This is not a restriction, however, because tuples $\sigma_1 \times \ldots \times \sigma_n \to \sigma$ can be considered as curried basic arrow-types $\sigma_1 \to \ldots \to \sigma_n \to \sigma$. Data-types and variable-types are lifted to expression-types via the rules,

$$\frac{\Gamma \vdash v : \tau}{\Gamma \vdash v : \mathbf{exp}[\tau]} \qquad \text{and} \qquad \frac{\Gamma \vdash \iota : \mathbf{var}[\tau]}{\Gamma \vdash !\iota : \mathbf{exp}[\tau]} \,.$$

Variables can be declared on data-types only, whereas *procedure-definition*, *recursion* and *conditional* are uniformly applicable to all phrase-types. An *environment* Γ is a partial function from *identifiers* to types, with *domain* $\mathrm{dom}(\Gamma)$.

The syntax is defined according to [Bro96], see Table 4.1 for an overview. Inaction is implemented by a **skip**-command; further, there are assignments ($V := E$), sequential ($C_1; C_2$) and parallel ($C_1 \parallel C_2$) composition, conditionals (**if** B **then** P_1 **else** P_2), iteration (**while** B **do** C), declaration (**new** $[\tau]\iota := E$ **in** C), atomicity (**await** B **then** C), as well as a call-by-name λ-calculus with recursion. As a slight extension, we allow for the use of conditionals in the body of **await**-statements. The body of an **await**-statement therefore consists of assignments,

$$\Gamma \vdash v : \tau$$

$$\Gamma \vdash \iota : \mathbf{var}[\tau] \quad \text{when } \Gamma(\iota) = \mathbf{var}[\tau]$$

$$\frac{\Gamma \vdash E_1 : \mathbf{exp}[\tau] \quad \Gamma \vdash E_2 : \mathbf{exp}[\tau]}{\Gamma \vdash E_1 \otimes E_2 : \mathbf{exp}[\tau]} \otimes : \tau \times \tau \to \tau$$

$$\Gamma \vdash \mathbf{skip} : \mathbf{comm}$$

$$\frac{\Gamma \vdash V : \mathbf{var}[\tau] \quad \Gamma \vdash E : \mathbf{exp}[\tau]}{\Gamma \vdash V := E : \mathbf{comm}}$$

$$\frac{\Gamma \vdash C_1 : \mathbf{comm} \quad \Gamma \vdash C_2 : \mathbf{comm}}{\Gamma \vdash C_1; C_2 : \mathbf{comm}}$$

$$\frac{\Gamma \vdash C_1 : \mathbf{comm} \quad \Gamma \vdash C_2 : \mathbf{comm}}{\Gamma \vdash C_1 \parallel C_2 : \mathbf{comm}}$$

$$\frac{\Gamma \vdash B : \mathbf{exp}[bool] \quad \Gamma \vdash P_1 : \theta \quad \Gamma \vdash P_2 : \theta}{\Gamma \vdash \mathbf{if}\ B\ \mathbf{then}\ P_1\ \mathbf{else}\ P_2 : \theta}$$

$$\frac{\Gamma \vdash B : \mathbf{exp}[bool] \quad \Gamma \vdash C : \mathbf{comm}}{\Gamma \vdash \mathbf{while}\ B\ \mathbf{do}\ C : \mathbf{comm}}$$

$$\frac{\Gamma \vdash B : \mathbf{exp}[bool] \quad \Gamma \vdash C : \mathbf{comm}}{\Gamma \vdash \mathbf{await}\ B\ \mathbf{then}\ C : \mathbf{comm}}$$

$$\frac{\Gamma \vdash E : \mathbf{exp}[\tau] \quad \Gamma, \iota : \mathbf{var}[\tau] \vdash C : \mathbf{comm}}{\Gamma \vdash \mathbf{new}\ [\tau]\iota := E\ \mathbf{in}\ C : \mathbf{comm}}$$

$$\Gamma \vdash x : \theta \quad \text{when } \Gamma(x) = \theta$$

$$\frac{\Gamma, x : \theta \vdash P : \theta}{\Gamma \vdash \mathbf{rec}\, x.P : \theta}$$

$$\frac{\Gamma, x : \theta \vdash P : \theta'}{\Gamma \vdash \lambda(x : \theta).P : (\theta \to \theta')}$$

$$\frac{\Gamma \vdash P_1 : (\theta \to \theta') \quad \Gamma \vdash P_2 : \theta}{\Gamma \vdash P_1(P_2) : \theta'}$$

Table 4.1: **Syntax and typing of CIA.** We follow standard syntax and typing conventions for Algol, and in particular CIA. The command in the body of an **await**-statement is a sequential composition of assignments and conditionals.

sequential composition and conditionals. This is important for the treatment of more involved examples, as in Section 4.3.1. Syntax and typing rules are presented in Table 4.1. Unlike Brookes, who considers iteration as a special kind of recursion, we explicitely add a **while**-construct to the language.

Semantics We define an SOS-style operational semantics for CIA, using small-step transition rules (as opposed to a big-step or natural semantics), in order to capture the nondeterministic behaviour resulting from the interaction of phrases via shared variables. The rules are fully standard, with the exception of those needed to model the atomicity required by **await**. The semantics is defined inductively for phrases of variable, expression, and command type, and is parameterized over the set os global variables used by a program.

Transitions take place between *configurations* $\langle P, \sigma \rangle$, consisting of a phrase P of variable, expression, or command type, and an assignment σ representing the local *and* global store. The set Γ of global variables is contained in $\mathrm{dom}(\sigma)$, that is, $\Gamma \subseteq \mathrm{dom}(\sigma)$. The main attention focusses on command-configurations. For them, a transition is basically of either of the forms,

$$\Gamma \vdash \langle C, \sigma \rangle \xrightarrow{\alpha} \langle C', \sigma' \rangle \qquad\qquad \Gamma \vdash \langle C, \sigma \rangle \xrightarrow{\surd} \sigma',$$

where $\alpha = \tau$ for each step of the program itself, and $\alpha \in \{\mathbf{out}_\iota(v), \overline{\mathbf{in}}_\iota(v)\}$ when the context reads from or writes to one of the global variables in σ. The 'tick' $\surd$ signifies termination, and σ' is the resulting state of the variables.

Structural equivalence We defer α-conversion and β-reduction to a standard *structural equivalence*, for which we write $\equiv_\alpha$. It deals with renamings of bound variables in order to avoid name-clashes (α-conversion), as well as with instantiations of λ-abstractions and unfoldings of recursive definitions (β-reduction). Structural congruence makes substitutions implicit which are usually necessary to implement α-conversion and β-reduction. In the soundness-proofs in Section 4.2, we are however interested in making these substitutions explicit. Hence, we use configurations of the form $\langle P\varrho\beta, \sigma \rangle$. There, ϱ contains the substitutions resulting from unfolding recursive definitions: $P\{\mathbf{rec}\, x.P/x\}$ instead of $\mathbf{rec}\, x.P$. And β contains those resulting from instantiations of λ-terms: $P\{P'/x\}$ for $(\lambda x.\, P)P'$.

In the following, we present operational semantics for CIA-$\{$**await**$\}$ (Section 4.1.1) and full CIA (Section 4.1.2), and introduce a behavioural congruence for CIA (Section 4.1.3).

4.1.1 CIA-{await}

The semantics of CIA-$\{$**await**$\}$ is fully standard. We define it on three levels: one for variables (Tables 4.2 and 4.4), one for expressions (Table 4.2 and 4.4), and a third one for commands (Table 4.3 and 4.4). There are no explicit rules for λ-abstraction and recursion. These are dealt with by the above call-by-name structural equivalence, $\equiv_\alpha$, covering α-conversion and β-reduction.

Variables and Expressions With CIA using passive expressions, the state is not modified by their execution. Expression-configurations $\langle E, \sigma \rangle$ with global variables Γ either perform

$$\frac{}{\Gamma \vdash \langle \iota, \sigma \rangle \xrightarrow{\surd} \iota}\ \mathbf{V1}$$

$$\frac{}{\Gamma \vdash \langle v, \sigma \rangle \xrightarrow{\surd} v}\ \mathbf{E1} \qquad \frac{\Gamma \vdash \langle V, \sigma \rangle \xrightarrow{\tau} \langle V', \sigma \rangle}{\Gamma \vdash \langle !V, \sigma \rangle \xrightarrow{\tau} \langle !V', \sigma \rangle}\ \mathbf{E2} \qquad \frac{\Gamma \vdash \langle V, \sigma \rangle \xrightarrow{\surd} \iota}{\Gamma \vdash \langle !V, \sigma \rangle \xrightarrow{\surd} \sigma(\iota)}\ \mathbf{E3}$$

$$\frac{\Gamma \vdash \langle E_1, \sigma \rangle \xrightarrow{\tau} \langle E_1', \sigma \rangle}{\Gamma \vdash \langle E_1 \otimes E_2, \sigma \rangle \xrightarrow{\tau} \langle E_1' \otimes E_2, \sigma \rangle}\ \mathbf{E4a} \qquad \frac{\Gamma \vdash \langle E_1, \sigma \rangle \xrightarrow{\surd} v_1 \qquad \Gamma \vdash \langle E_2, \sigma \rangle \xrightarrow{\surd} v_2}{\Gamma \vdash \langle E_1 \otimes E_2, \sigma \rangle \xrightarrow{\surd} v_1 \otimes v_2}\ \mathbf{E5}$$

Table 4.2: **Operational semantics for CIA: Variables and Expressions**. The evaluation of a variable or an expression cannot change the state, hence σ is never modified. We omit rule **E4b** which is a symmetric version of rule **E4a**. The set $\Gamma \subseteq \mathrm{dom}(\sigma)$ contains the global variables; that is, those variables that can be read and modified by a context.

silent evaluation-steps, which can include reading from a variable in σ, or they terminate producing a value v:

$$\Gamma \vdash \langle E, \sigma \rangle \xrightarrow{\tau} \langle E', \sigma' \rangle \qquad\qquad \Gamma \vdash \langle E, \sigma \rangle \xrightarrow{\surd} v.$$

Again, $\Gamma \subseteq \mathrm{dom}(\sigma)$ denotes the set of global variables. Table 4.2 gives the basic transition-rules for variable- (**V1**) and expression-configurations (**E1**–**E5**). These are extended by the rules in Table 4.4, which apply to all phrase-types, and are therefore added only once.

Remark: Note that there is no interaction with the environment in these rules. This only happens on the level of commands. In fact, observational equivalence is defined on the level of command-configurations only—see Definition 4.1 below—and is then extended to arbitrary phrase-types by a closure argument in Definition 4.2.

Commands As stated above, transitions of command-configurations are of the form,

$$\Gamma \vdash \langle C, \sigma \rangle \xrightarrow{\alpha} \langle C', \sigma' \rangle \qquad\qquad \Gamma \vdash \langle C, \sigma \rangle \xrightarrow{\surd} \sigma',$$

where $\alpha = \tau$ for each step of the program itself, and $\alpha \in \{\mathbf{out}_\iota(v), \overline{\mathbf{in}}_\iota(v)\}$ when the context reads from or writes to one of the global variables in σ. This interaction of a context with the global variables is described by rules **C11** and **C12** in Table 4.3.

A declaration of a local variable adds a new cell to σ (**C4**–**C5**); α-conversion guarantees that a fresh cell can be created whenever necessary.

4.1.2 Full CIA

Parallel languages need a form of *atomicity*, so that mutual exclusion can be implemented. In CIA, atomicity is present in terms of an **await**-command: during its execution, the whole context is blocked. This includes that even the environment cannot access the global variables during that time. The semantics of a command **await** B **then** C with boolean expression B and command C is as follows: First B is evaluated; during that time already, the context is blocked. If B evaluates to *true* (*tt*), command C is executed immediately, without unblocking

$$\frac{}{\Gamma \vdash \langle \mathbf{skip}, \sigma \rangle \xrightarrow{\surd} \sigma}\ \mathbf{C1} \qquad \frac{\Gamma \vdash \langle V, \sigma \rangle \xrightarrow{\surd} \iota \quad \Gamma \vdash \langle E, \sigma \rangle \xrightarrow{\surd} v}{\Gamma \vdash \langle V := E, \sigma \rangle \xrightarrow{\surd} \sigma[\iota := v]}\ \mathbf{C2}$$

$$\frac{\Gamma \vdash \langle V, \sigma \rangle \xrightarrow{\tau} \langle V', \sigma \rangle}{\Gamma \vdash \langle V := E, \sigma \rangle \xrightarrow{\tau} \langle V' := E, \sigma \rangle}\ \mathbf{C3a} \qquad \frac{\Gamma \vdash \langle E, \sigma \rangle \xrightarrow{\tau} \langle E', \sigma \rangle}{\Gamma \vdash \langle V := E, \sigma \rangle \xrightarrow{\tau} \langle V := E', \sigma \rangle}\ \mathbf{C3b}$$

$$\frac{\Gamma \vdash \langle E, \sigma \rangle \xrightarrow{\tau} \langle E', \sigma \rangle}{\Gamma \vdash \langle \mathbf{new}\ [\tau]\iota := E\ \mathbf{in}\ C, \sigma \rangle \xrightarrow{\tau} \langle \mathbf{new}\ [\tau]\iota := E'\ \mathbf{in}\ C, \sigma \rangle}\ \mathbf{C4}$$

$$\frac{\Gamma \vdash \langle E, \sigma \rangle \xrightarrow{\surd} v \qquad \iota \notin \mathrm{dom}(\sigma)}{\Gamma \vdash \langle \mathbf{new}\ [\tau]\iota := E\ \mathbf{in}\ C, \sigma \rangle \xrightarrow{\tau} \langle C, \sigma \cup \{(\iota, v)\} \rangle}\ \mathbf{C5}$$

$$\frac{\Gamma \vdash \langle C_1, \sigma \rangle \xrightarrow{\mu} \langle C_1', \sigma' \rangle}{\Gamma \vdash \langle C_1; C_2, \sigma \rangle \xrightarrow{\mu} \langle C_1'; C_2, \sigma' \rangle}\ \mathbf{C6} \qquad \frac{\Gamma \vdash \langle C_1, \sigma \rangle \xrightarrow{\surd} \sigma'}{\Gamma \vdash \langle C_1; C_2, \sigma \rangle \xrightarrow{\tau} \langle C_2, \sigma' \rangle}\ \mathbf{C7}$$

$$\frac{\Gamma \vdash \langle C_1, \sigma \rangle \xrightarrow{\mu} \langle C_1', \sigma' \rangle}{\Gamma \vdash \langle C_1 \parallel C_2, \sigma \rangle \xrightarrow{\mu} \langle C_1' \parallel C_2, \sigma' \rangle}\ \mathbf{C8a} \qquad \frac{\Gamma \vdash \langle C_1, \sigma \rangle \xrightarrow{\surd} \sigma'}{\Gamma \vdash \langle C_1 \parallel C_2, \sigma \rangle \xrightarrow{\tau} \langle C_2, \sigma' \rangle}\ \mathbf{C9a}$$

$$\frac{}{\Gamma \vdash \langle \mathbf{while}\ B\ \mathbf{do}\ C, \sigma \rangle \xrightarrow{\tau} \langle \mathbf{if}\ B\ \mathbf{then}\ (C; \mathbf{while}\ B\ \mathbf{do}\ C)\ \mathbf{else\ skip}, \sigma \rangle}\ \mathbf{C10}$$

$$\frac{\iota \in \Gamma}{\Gamma \vdash \langle C, \sigma \rangle \xrightarrow{\overline{\mathbf{in}}_\iota \langle \sigma(\iota) \rangle} \langle C, \sigma \rangle}\ \mathbf{C11} \qquad \frac{\iota \in \Gamma}{\Gamma \vdash \langle C, \sigma \rangle \xrightarrow{\mathbf{out}_\iota(v)} \langle C, \sigma[\iota := v] \rangle}\ \mathbf{C12}$$

Table 4.3: **Operational semantics for CIA-{await}: Commands**. The execution of commands can affect the state, which means that σ can be modified. We omit rules **C8b** and **C9b** which are symmetric versions of rules **C8a** and **C9a**. The set $\Gamma \subseteq \mathrm{dom}(\sigma)$ contains the global variables; that is, those variables that can be read and modified by a context. The last two rules, **C11** and **C12** give the context access to the global variables.

$$\frac{\Gamma \vdash \langle B, \sigma \rangle \xrightarrow{\tau} \langle B', \sigma \rangle}{\Gamma \vdash \langle \mathbf{if}\ B\ \mathbf{then}\ P_1\ \mathbf{else}\ P_2, \sigma \rangle \xrightarrow{\tau} \langle \mathbf{if}\ B'\ \mathbf{then}\ P_1\ \mathbf{else}\ P_2, \sigma \rangle}\ \mathbf{P1}$$

$$\frac{\Gamma \vdash \langle B, \sigma \rangle \xrightarrow{\surd} tt}{\Gamma \vdash \langle \mathbf{if}\ B\ \mathbf{then}\ P_1\ \mathbf{else}\ P_2, \sigma \rangle \xrightarrow{\tau} \langle P_1, \sigma \rangle}\ \mathbf{P2} \qquad \frac{\Gamma \vdash \langle B, \sigma \rangle \xrightarrow{\surd} f\!f}{\Gamma \vdash \langle \mathbf{if}\ B\ \mathbf{then}\ P_1\ \mathbf{else}\ P_2, \sigma \rangle \xrightarrow{\tau} \langle P_2, \sigma \rangle}\ \mathbf{P3}$$

$$\frac{\Gamma \vdash P \equiv_\alpha P' \qquad \Gamma \vdash \langle P', \sigma \rangle \xrightarrow{\mu} \mathcal{C}}{\Gamma \vdash \langle P, \sigma \rangle \xrightarrow{\mu} \mathcal{C}}\ \mathbf{P4}$$

Table 4.4: **Operational semantics for CIA-{await}: Phrases**. The only phrases which we consider directly, are conditionals and recursive phrases of variable, expression, or command type. Conditionals are evaluated in the usual way. Function application and recursion are not considered explicitely; β-reduction is applied implicitely, according to the usual scheme. The set $\Gamma \subseteq \mathrm{dom}(\sigma)$ contains the global variables; that is, those variables that can be read and modified by a context. $\mathcal{C}$ denotes a configuration or a result, and $\mu \in \{\tau, \overline{\mathbf{in}}_\iota\langle v \rangle, \mathbf{out}_\iota(v)\}$.

$$\frac{}{\Gamma \vdash \langle \mathbf{await}\ B\ \mathbf{then}\ C, \sigma\rangle \xrightarrow{\tau} \langle \mathcal{A}[\![B]\!]^B_C, \sigma\rangle_\ell}\ \textbf{Cl1} \qquad \frac{\Gamma \vdash \langle B', \sigma\rangle \xrightarrow{\tau} \langle B'', \sigma\rangle}{\Gamma \vdash \langle \mathcal{A}[\![B']\!]^B_C, \sigma\rangle_\ell \xrightarrow{\tau} \langle \mathcal{A}[\![B'']\!]^B_C, \sigma\rangle_\ell}\ \textbf{Cl2}$$

$$\frac{\Gamma \vdash \langle B', \sigma\rangle \xrightarrow{\surd} tt}{\Gamma \vdash \langle \mathcal{A}[\![B']\!]^B_C, \sigma\rangle_\ell \xrightarrow{\tau} \langle \mathcal{A}[\![C]\!], \sigma\rangle_\ell}\ \textbf{Cl3a} \qquad \frac{\Gamma \vdash \langle B', \sigma\rangle \xrightarrow{\surd} \mathit{ff}}{\Gamma \vdash \langle \mathcal{A}[\![B']\!]^B_C, \sigma\rangle_\ell \xrightarrow{\tau} \langle \mathbf{await}\ B\ \mathbf{then}\ C, \sigma\rangle}\ \textbf{Cl3b}$$

$$\frac{\Gamma \vdash \langle C, \sigma\rangle \xrightarrow{\tau} \langle C', \sigma'\rangle}{\Gamma \vdash \langle \mathcal{A}[\![C]\!], \sigma\rangle_\ell \xrightarrow{\tau} \langle \mathcal{A}[\![C']\!], \sigma'\rangle_\ell}\ \textbf{Cl4} \qquad \frac{\Gamma \vdash \langle C, \sigma\rangle \xrightarrow{\surd} \sigma'}{\Gamma \vdash \langle \mathcal{A}[\![C]\!], \sigma\rangle_\ell \xrightarrow{\surd} \sigma'}\ \textbf{Cl5}$$

$$\frac{\Gamma \vdash \langle C_1, \sigma\rangle_\zeta \xrightarrow{\mu} \langle C_1', \sigma'\rangle_{\zeta'}}{\Gamma \vdash \langle C_1; C_2, \sigma\rangle_\zeta \xrightarrow{\mu} \langle C_1'; C_2, \sigma'\rangle_{\zeta'}}\ \textbf{C6'} \qquad \frac{\Gamma \vdash \langle C_1, \sigma\rangle_\zeta \xrightarrow{\surd} \sigma'}{\Gamma \vdash \langle C_1; C_2, \sigma\rangle_\zeta \xrightarrow{\tau} \langle C_2, \sigma'\rangle}\ \textbf{C7'}$$

$$\frac{\Gamma \vdash \langle C_1, \sigma\rangle_\zeta \xrightarrow{\mu} \langle C_1', \sigma'\rangle_{\zeta'}}{\Gamma \vdash \langle C_1 \parallel C_2, \sigma\rangle_\zeta \xrightarrow{\mu} \langle C_1' \parallel C_2, \sigma'\rangle_{\zeta'}}\ \textbf{C8a'} \qquad \frac{\Gamma \vdash \langle C_1, \sigma\rangle_\zeta \xrightarrow{\surd} \sigma'}{\Gamma \vdash \langle C_1 \parallel C_2, \sigma\rangle_\zeta \xrightarrow{\tau} \langle C_2, \sigma'\rangle}\ \textbf{C9a'}$$

$$\frac{\Gamma \vdash C \equiv_\alpha C' \qquad \Gamma \vdash \langle C', \sigma\rangle_\zeta \xrightarrow{\mu} \mathcal{C}}{\Gamma \vdash \langle C, \sigma\rangle_\zeta \xrightarrow{\mu} \mathcal{C}}\ \textbf{P4'}$$

Table 4.5: **Operational semantics for Full CIA: Commands**. We state additional rules for introducing and removing locks. Whenever a configuration is marked with a lock, only that component which has introduced it can compute; all others have to wait until the lock has been released again. The subscripts ζ and ζ' denote ℓ for locked configurations and ϵ for unlocked ones.

the context in-between. If B evaluates to *false* (ff), the context is released, and the command repeats the evaluation of B after a period of busy-waiting.

In our operational semantics, this is achieved by introducing *locked configurations* $\langle P, \sigma\rangle_\ell$. The tag ℓ represents a lock. Whenever an **await**-statement is executed, the configuration is marked with the lock ℓ, and all but the **await**-component are prevented from running (this component is marked itself with an $\mathcal{A}[\![.]\!]$-construct, so to be distinguishable from its context). The lock is released either if the guarding boolean expression has been evaluated to false or, otherwise, after the command has been completed.

Variables and Expressions As locks only affect command-configurations, the operational semantics of variables and expressions is exactly the same as for CIA-{**await**}. This means that the rules from Tables 4.2 and 4.4 work for variable- and expression-configurations for full CIA as well. Note that there are no locked such configurations.

Commands The rules for locked command-configurations are of the form

$$\Gamma \vdash \langle C, \sigma\rangle_\zeta \xrightarrow{\alpha} \langle C', \sigma'\rangle_{\zeta'} \qquad\qquad \Gamma \vdash \langle C, \sigma\rangle_\zeta \xrightarrow{\surd} \sigma',$$

where ζ and ζ' represent the presence or absence of a lock. Table 4.5 gives rules for introducing (**Cl1**) and removing (**Cl3b** and **Cl5**) locks; in general, there are rules for executing the **await**-statement (**Cl1**–**Cl5**), as well as locked variations of the rules dealing with sequential (**C6'** and **C7'**) and parallel (**C8a'**–**C9b'**) composition. The other rules are as before, see Tables 4.3 and 4.4.

Remark: Note that a locked configuration cannot perform a visible transition. As a consequence, the environment is blocked from accessing the global variables during the execution of an **await**-statement, as is the program-context.

4.1.3 Observational Congruence

We relate CIA-phrases of corresponding types by means of a *behavioural congruence*, based on an equivalence over *closed commands*. A command C is Γ-*closed*, if its only free identifiers in Γ are of variable-type; that is, $\Gamma = \iota_1 : \mathbf{var}[\tau_1], \ldots, \iota_n : \mathbf{var}[\tau_n]$ Like this, $\iota_1, \ldots, \iota_n$ can be taken care of by a state σ. Note that all command-configurations from the previous sections employed closed commands.

Weak transitions We abstract from internal activities by using *weak transitions* $\mathcal{C} \stackrel{\hat{\mu}}{\Longrightarrow} \mathcal{C}$ between configurations. The relation $\stackrel{\epsilon}{\Longrightarrow}$ is the reflexive and transitive closure of $\stackrel{\tau}{\longrightarrow}$, and $\stackrel{\hat{\mu}}{\Longrightarrow}$ is given by $\stackrel{\epsilon}{\Longrightarrow}\stackrel{\mu}{\longrightarrow}\stackrel{\epsilon}{\Longrightarrow}$ (arbitrarily many invisible steps before and after the μ transition) for visible $\mu \in \{\mathbf{out}_\iota(v), \overline{\mathbf{in}_\iota}\langle v\rangle\}$, and $\stackrel{\epsilon}{\Longrightarrow}$ for an invisible $\mu = \tau$. Note that we use exactly the same definition of weak transitions as in CCS or the π-calculus, see Chapter 2.

Behavioural equivalence process calculi to CIA command-configurations. We use an early semantics, because we consider it more natural for languages of the kind of CIA to instantiate input-objects as soon as they interact with the environment; that is, when writing on a global variable, the environment immediately provides a value.

DEFINITION 4.1 (CONFIGURATION-BISIMULATION) A binary relation $\mathcal{R}(\Gamma)$ upon command-configurations is a *configuration bisimulation* if $\mathcal{C}_1\mathcal{R}(\Gamma)\mathcal{C}_2$ implies (where the free variables in $\mathcal{C}_1$ and $\mathcal{C}_2$ are captured by Γ),

1. $\Gamma \vdash \mathcal{C}_1 \stackrel{\surd}{\longrightarrow} \sigma_1$ implies there is a σ_2 such that $\Gamma \vdash \mathcal{C}_2 \stackrel{\surd}{\Longrightarrow} \sigma_2$, where $\sigma_1(\iota) = \sigma_2(\iota)$ for all $\iota \in \Gamma$;
2. $\Gamma \vdash \mathcal{C}_1 \stackrel{\mu}{\longrightarrow} \mathcal{C}_1'$ implies there is a $\mathcal{C}_2'$ such that $\Gamma \vdash \mathcal{C}_2 \stackrel{\hat{\mu}}{\Longrightarrow} \mathcal{C}_2'$ and $\mathcal{C}_1'\mathcal{R}(\Gamma)\mathcal{C}_2'$, where $\mu \in \{\tau, \mathbf{out}_\iota(v), \overline{\mathbf{in}_\iota}\langle v\rangle\}$;
3. symmetrically for transitions of $\mathcal{C}_2$.

We write $\Gamma \vdash \mathcal{C}_1 \approx \mathcal{C}_2$ if there is a configuration bisimulation $\mathcal{R}(\Gamma)$ with $\mathcal{C}_1\mathcal{R}(\Gamma)\mathcal{C}_2$.

Observational congruence We say that a context Con is Γ-*closed with respect to* a phrase P if $\Gamma \vdash Con[P] : \mathbf{comm}$ for some $\Gamma = \iota_1 : \mathbf{var}[\tau_1], \ldots, \iota_n : \mathbf{var}[\tau_n]$; that is, if $Con[P]$ does not contain free identifiers except for global variables $\iota_1, \ldots, \iota_n$.

With this notion of closure, we are now able to define a behavioural congruence for arbitrary phrases. The idea behind this definition is the following: we would like to identify exactly those phrases that cannot be distinguished by any context yielding a closed command.

DEFINITION 4.2 (OBSERVATIONAL CONGRUENCE) Let P_1, P_2 be arbitrary phrases with identifiers in Γ. Then P_1 and P_2 are *observationally congruent*, written $\Gamma \vdash P_1 \approx_{oc} P_2$, if for every

context Con which is Γ'-closed with respect to P_1 and P_2 such that $\Gamma' \subseteq \Gamma$, and every σ with $\mathrm{dom}(\sigma) = \Gamma'$, it holds that $\Gamma' \vdash \langle Con[P_1], \sigma\rangle \approx \langle Con[P_2], \sigma\rangle$.

Observational congruence is the notion of behavioural equality on CIA-phrases we are interested in. It is however hard to prove equalities following its definition, due to the universal quantification over the contexts and assignments.

Determinacy of await-statements We conclude the section with a useful fact about locked configurations. The behaviour of an **await**-statement is deterministic, both due to the absence of parallel composition within its body and the incapability of expressions to change a given assignment. Therefore, locked configurations either diverge (during the evaluation of their guarding boolean expression) or yield bisimilar configurations without a lock (after their execution). This allows us to abstract from locked configurations in the full-abstraction proof of Theorem 4.13 in Section 4.2.2.

LEMMA 4.3 $\Gamma \vdash \langle C, \sigma\rangle_\ell \xrightarrow{\tau} \langle C', \sigma'\rangle_\zeta$ *with* $\zeta \in \{\ell, \epsilon\}$ *implies* $\Gamma \vdash \langle C, \sigma\rangle_\ell \approx \langle C', \sigma'\rangle_\zeta$.

Proof: By rule-induction. Only few rules are applicable for locked configurations, see Table 4.5. These rules yield a fully deterministic execution, because the environment is blocked by the lock too. □

COROLLARY 4.4 *For every configuration* $\langle C, \sigma\rangle_\ell$ *and* $\Gamma \subseteq dom(\sigma)$ *the following holds: Either it diverges (that is, there is an infinite computation of silent steps starting from* $\langle C, \sigma\rangle_\ell$*) or there is another configuration* $\langle C', \sigma'\rangle$ *such that* $\Gamma \vdash \langle C, \sigma\rangle_\ell \stackrel{\tau}{\Longrightarrow} \langle C', \sigma'\rangle$ *and* $\Gamma \vdash \langle C, \sigma\rangle_\ell \approx \langle C', \sigma'\rangle$.

4.2 Encoding Concurrent Idealized Algol

Recall from Section 4.1 that during a run of a program with explicit substitutions, configurations $\langle P, \sigma\rangle$ with phrase P and store σ are extended with substitutions ϱ and β, capturing recursive phrases and function-applications, respectively; hence, a general notion of configuration can be described by $\langle P\varrho\beta, \sigma\rangle$. We encode each of these elements in the π-calculus.

Encoding phrases For the purpose of encoding arbitrary CIA-phrases, we combine traditional encodings of variables and commands, as in [Mil89], for instance, with a translation of the call-by-name λ-calculus, as proposed in [Mil92b]. Depending on whether the phrases contain **await**-statements or not, we use one of the encodings from Tables 4.6 or 4.7. The encoding $[\![P]\!]_p$ of a CIA-phrase P is parameterized over a name p which is used to signal termination. As a simple example, the command **skip** is encoded in terms of a process $[\![\mathbf{skip}]\!]_p = \bar{p}.0$; a value v yields $[\![v]\!]_p = \bar{p}\langle v\rangle.0$, and a variable ι is translated into $[\![\iota]\!]_p = \bar{p}\langle \mathbf{in}_\iota, \mathbf{out}_\iota\rangle.0$. Note that this encoding of variables implements the idea of identifying a variable with its two methods for reading and writing [Rey81, AM96a, AM99]. We present the two encodings in more detail below, in two separate sections for CIA-{**await**} (Section 4.2.1) and full CIA (Section 4.2.2).

Encoding state An assignment σ consists of pairs of reference-cells and values stored into them. We model these cells ι explicitely, in terms of recursive agents of the form (recall from Section 2.2.3 that recursion can be implemented by replication),

$$\mathbf{Reg}_\iota[v] \stackrel{\text{def}}{=} \overline{\mathbf{in}}_\iota\langle v\rangle.\mathbf{Reg}_\iota[v] \;+\; \mathbf{out}_\iota(w).\mathbf{Reg}_\iota[w].$$

Processes in the scope of $fn_\iota \stackrel{\text{def}}{=} \{\mathbf{in}_\iota, \mathbf{out}_\iota\}$ are allowed to read from and write to $\mathbf{Reg}_\iota$.

Recall that we distinguish between local and global variables. Thus, when encoding a configuration $\langle P, \sigma \rangle$ with global variables in $\Gamma \subseteq \mathrm{dom}(\sigma)$, we put a restriction only over those fn_ι for which $\iota \in \mathrm{dom}(\sigma) - \Gamma$, that is, over the local variables. In order to restrict the environment to communications with the global variables, that is, to prevent access to those parts of P trying to read from or write to the respective registers, we employ the I/O-type system by Pierce and Sangiorgi [PS96] introduced in Section 2.2.3.

Encoding substitutions Recall from Section 4.1 that when making substitutions explicit in configurations, we can distinguish between those dealing with recursions (which we denote by ϱ) and those for function-applications (for which we write β).

A substitution $\mathbf{rec}\, x_i.P_i^\varrho/x_i$ from $\varrho = \{\mathbf{rec}\, x_1.P_1^\varrho/x_1, \ldots, \mathbf{rec}\, x_m.P_m^\varrho/x_m\}$ is modelled by a replication of the encoding of phrase P_i^ϱ accessible via x_i, that is, by $!x_i(r).[\![P_i^\varrho]\!]_r$ with parameter q. Whenever x_i occurs in the body of P_i^ϱ during its execution, a copy of $[\![P_i^\varrho]\!]_q$ can thus be released. This is a standard way of encoding recursion in the π-calculus.

A substitution P_i^β/y_i from $\beta = \{P_1^\beta/y_1, \ldots, P_n^\beta/y_n\}$ is modelled similarly, by a replication of the encoding of P_i^β accessible via y_i, that is, by $!y_i(r).[\![P_i^\beta]\!]_r$. Whenever, y_i occurs in the body of the λ-abstraction (which is not part of the substitution) during its execution, a copy of the argument P_i^β is released. This way of encoding function application in a call-by-name λ-calculus has been proposed by Milner in [Mil92b].

Encoding configurations We are now ready to define translations of configurations into the π-calculus. These encodings are given in terms of a parallel composition of the elements (phrase, substitutions, and assignment) together with suitable restrictions over the names to access the substitutions, and local variables.

A configuration $\langle P\varrho\beta, \sigma \rangle$ with substitutions $\varrho = \{\mathbf{rec}\, x_1.P_1^\varrho/x_1, \ldots, \mathbf{rec}\, x_m.P_m^\varrho/x_m\}$ and $\beta = \{P_1^\beta/y_1, \ldots, P_n^\beta/y_n\}$, local variables $\mathrm{dom}(\sigma) - \Gamma = \{\iota_1, \ldots, \iota_l\}$, and global variables $\Gamma = \{\kappa_1, \ldots, \kappa_k\}$, translates to the following process (recall from above that $fn_\iota = \{\mathbf{in}_\iota, \mathbf{out}_\iota\}$):

$(\nu\, x_1, \ldots, x_m, y_1, \ldots, y_n, fn_{\iota_1}, \ldots, fn_{\iota_l})$	restrictions establishing scope of $\varrho, \beta, \mathrm{dom}(\sigma) - \Gamma$
$(!x_1(r).[\![P_1^\varrho]\!]_r \mid \ldots \mid !x_m(r).[\![P_m^\varrho]\!]_r \mid$	substitutions from recursions ϱ
$!y_1(r).[\![P_1^\beta]\!]_r \mid \ldots \mid !y_n(r).[\![P_n^\beta]\!]_r \mid$	substitutions from function applications β
$\mathbf{Reg}_{\iota_1}[\sigma(\iota_1)] \mid \ldots \mid \mathbf{Reg}_{\iota_l}[\sigma(\iota_l)] \mid$	local variables in $\mathrm{dom}(\sigma) - \Gamma$
$\mathbf{Reg}_{\kappa_1}[\sigma(\kappa_1)] \mid \ldots \mid \mathbf{Reg}_{\kappa_k}[\sigma(\kappa_k)] \mid$	global variables in Γ
$[\![P]\!]_p)$	encoding of phrase P

We write $[\![\Gamma \vdash \langle P\varrho\beta, \sigma \rangle]\!]_p$ for encodings of configurations with global variables in Γ.

Remark: The translations of ϱ and β show how close the relationship is between recursion and λ-abstraction. In fact, recursion can be considered as a generalization of λ-abstraction with self-application, where the argument is itself in the scope of the λ-abstraction. This becomes particularly obvious in the π-calculus, because it has a first-order syntax, hence references are used to access both recursive agents and function-arguments.

4.2.1 Encoding CIA-{await}

As noted above, we translate all phrases P into parameterized processes $[\![P]\!]_p$ with the fresh name p being used to signal the termination of the execution of $[\![P]\!]_p$. We will see below that

Variables, Expressions

$$
\begin{aligned}
[\![\iota]\!]_p &\stackrel{\text{def}}{=} \bar{p}\langle \mathbf{in}_\iota, \mathbf{out}_\iota \rangle.0 \\
[\![v]\!]_p &\stackrel{\text{def}}{=} \bar{p}\langle v \rangle.0 \\
[\![!V]\!]_p &\stackrel{\text{def}}{=} (\nu\, q)([\![V]\!]_q \mid q(i,o).i(x).\bar{p}\langle x \rangle.0) \\
[\![E_1 \otimes E_2]\!]_p &\stackrel{\text{def}}{=} (\nu\, q,r)([\![E_1]\!]_q \mid [\![E_2]\!]_r \mid q(x).r(y).\overline{p}\langle x \otimes y \rangle.0)
\end{aligned}
$$

Commands

$$
\begin{aligned}
[\![\mathbf{skip}]\!]_p &\stackrel{\text{def}}{=} \bar{p}.0 \\
[\![V := E]\!]_p &\stackrel{\text{def}}{=} (\nu\, q,r)([\![V]\!]_q \mid [\![E]\!]_r \mid q(i,o).r(x).\bar{o}\langle x \rangle.\bar{p}.0) \\
[\![\mathbf{new}\ [\tau]\iota := E\ \mathbf{in}\ C]\!]_p &\stackrel{\text{def}}{=} (\nu\, q)([\![E]\!]_q \mid q(x).(\nu\, fn_\iota)(\mathbf{Reg}_\iota[x] \mid [\![C]\!]_p)) \\
[\![C_1; C_2]\!]_p &\stackrel{\text{def}}{=} (\nu\, q)([\![C_1]\!]_q \mid q.[\![C_2]\!]_p) \\
[\![C_1 \parallel C_2]\!]_p &\stackrel{\text{def}}{=} (\nu\, q,r)([\![C_1]\!]_q \mid [\![C_2]\!]_r \mid q.r.\bar{p}.0) \\
[\![\mathbf{while}\ B\ \mathbf{do}\ C]\!]_p &\stackrel{\text{def}}{=} (\nu\, a)(!a.\,(\nu\, q)([\![B]\!]_q \mid q(x).\,([x = tt]\,(\nu\, r)([\![C]\!]_r \mid r.\bar{a}.0) \mid \\
&\qquad\qquad\qquad\qquad\qquad [x = f\!f]\,\bar{p}.0)) \mid \bar{a}.0)
\end{aligned}
$$

Phrases

$$
\begin{aligned}
[\![\mathbf{if}\ B\ \mathbf{then}\ P_1\ \mathbf{else}\ P_2]\!]_p &\stackrel{\text{def}}{=} (\nu\, q)([\![B]\!]_q \mid q(x).([x = tt]\,[\![P_1]\!]_p \mid [x = f\!f]\,[\![P_2]\!]_p)) \\
[\![x]\!]_p &\stackrel{\text{def}}{=} \bar{x}\langle p \rangle.0 \\
[\![\mathbf{rec}\, x.\, P]\!]_p &\stackrel{\text{def}}{=} (\nu\, x)(!x(r).[\![P]\!]_r \mid \bar{x}\langle p \rangle.0) \\
[\![\lambda(x : \theta).\, P]\!]_p &\stackrel{\text{def}}{=} (\nu\, v)(\bar{p}\langle v \rangle.v(x,q).[\![P]\!]_q) \\
[\![P_1 P_2]\!]_p &\stackrel{\text{def}}{=} (\nu\, q)([\![P_1]\!]_q \mid q(v).(\nu\, x)(\bar{v}\langle x,p \rangle.!x(r).[\![P_2]\!]_r))
\end{aligned}
$$

Table 4.6: **Encoding CIA-{await}.** Phrases P are translated into π-calculus processes $[\![P]\!]_p$ according to the above rules.

for P being an expression or a command, $[\![P]\!]_p$ produces an output on p, exactly if and only if P terminates in the operational semantics of CIA; this fact is one part of the *operational correspondence* which we use to establish that our encoding is sound.

Table 4.6 gives an overview of the translating rules. As a typical example, consider the sequential composition of two commands. Sequentiality is not a basic concept in the π-calculus, but it can be encoded by exploiting the parameter attached to each encoding:

$$[\![C_1; C_2]\!]_p \stackrel{\text{def}}{=} (\nu\, q)([\![C_1]\!]_q \mid q.[\![C_2]\!]_p).$$

First, only $[\![C_1]\!]_q$ is able to execute because of name q guarding $[\![C_2]\!]_p$. As soon as $[\![C_1]\!]_q$ terminates, it signals so on $\overline{q}$ and releases $[\![C_2]\!]_p$. Yet another example of sequentiality are declarations, in which an expression is evaluated, strictly before creating a new (private) location with the result, and executing the continuation,

$$[\![\mathbf{new}\ [\tau]\iota := E\ \mathbf{in}\ C]\!]_p \stackrel{\text{def}}{=} (\nu\, q)([\![E]\!]_q \mid q(x).(\nu\, fn_\iota)(\mathbf{Reg}_\iota[x] \mid [\![C]\!]_p)).$$

Here, parameter q does not only guard $[\![C]\!]_p$, but is also used to transmit the result of the evaluation of $[\![E]\!]_q$ to register $\mathbf{Reg}_\iota$. Suppose E is a value v, then,

$$\begin{aligned}[\![\mathbf{new}\ [\tau]\iota := v\ \mathbf{in}\ C]\!]_p &\stackrel{\text{def}}{=} (\nu\, q)(\overline{q}\langle v\rangle.0 \mid q(x).(\nu\, fn_\iota)(\mathbf{Reg}_\iota[x] \mid [\![C]\!]_p)) \\ &\approx_\pi (\nu\, fn_\iota)(\mathbf{Reg}_\iota[v] \mid [\![P]\!]_p),\end{aligned}$$

where $\approx_\pi$ is an application of some simple π-calculus laws; to be precise, of the value-transmission law $(\nu\, q)(\overline{q}\langle v\rangle.R \mid q(x).S) \approx_\pi (\nu\, q)(R \mid S\{v/x\})$, and of the garbage-collection law $(\nu\, q)R \approx_\pi R$ if q is not free in R (see also Section 2.2.2). Identifiers are modelled by processes sending along a specified channel which is used to invoke a copy of the argument they represent. Both procedural arguments and recursion are translated using replication, so fresh copies are available at every call (recall that CIA is a call-by-name language). For instance, if P is a free identifier, called x_P in the π-calculus translation, then

$$\begin{aligned}&[\![\mathbf{new}\ [int]\iota := 1\ \mathbf{in}\ P(!\iota)]\!]_p \\ &\approx_\pi \underbrace{(\nu\, fn_\iota)(\mathbf{Reg}_\iota[1]}_{\text{Declaration}} \mid \underbrace{(\nu\, q)(\overline{x_P}\langle q\rangle.0 \mid q(v)}_{\text{Invoking a copy of procedure } P}.\ \underbrace{(\nu\, x)(\overline{v}\langle x, p\rangle}_{\text{Communicating argument and termination signal}}\ .\underbrace{!x(r).\mathbf{in}_\iota(z).\overline{r}\langle z\rangle.0)))}_{\text{Procedural argument}}.\end{aligned}$$

An auxiliary encoding In the results concerning operational correspondence—see Lemmas 4.5 and 4.6 below—we make use of an auxiliary encoding of CIA phrases $[\![P]\!]^*_p$, which is essentially like $[\![P]\!]_p$ but with administrative steps already performed. For example, the usual encoding of the assignment of a value v to a variable ι given by,

$$[\![\iota := v]\!]_p = (\nu\, q, r)(\bar{q}\langle \mathbf{in}_\iota, \mathbf{out}_\iota\rangle.0 \mid \bar{r}\langle v\rangle.0 \mid q(i, o).r(x).\bar{o}\langle x\rangle.\bar{p}.0),$$

reduces to the following process, where the results of evaluating variable $[\![\iota]\!]_q$ and expression $[\![v]\!]_r$ have already been transmitted to the continuation,

$$[\![\iota := v]\!]^*_p = \overline{\mathbf{out}_\iota}\langle v\rangle.\bar{p}.0.$$

The auxiliary translation is chosen such that for all phrases P, the standard encoding expands it, that is, $[\![P]\!]_p \geq [\![P]\!]_p^*$. By compositionality of expansion, this extends to encodings of configurations as well, that is, $[\![\Gamma \vdash \langle P\varrho\beta,\sigma\rangle]\!]_p \geq [\![\Gamma \vdash \langle P\varrho\beta,\sigma\rangle]\!]_p^*$. The auxiliary encoding therefore does not encode single configurations, but classes of configurations which are behaviourally equivalent but can perform different administrative steps.

Remark: It is interesting to see that, when applying the auxiliary encoding to closed commands, the only relevant transitions are those in which a program reads from or writes to a variable. The reason is that the content of the variable can determine the future behaviour of a configuration, whereas all other steps only have administrative character.

Operational correspondence There is a close *operational correspondence* between configurations $\Gamma \vdash \langle P\varrho\beta,\sigma\rangle$ and their (auxiliary) encodings $[\![\Gamma \vdash \langle P\varrho\beta,\sigma\rangle]\!]_p^*$. For instance, as indicated above, a configuration $\Gamma \vdash \langle C\varrho\beta,\sigma\rangle$ over a command C terminates if and only if its translation $[\![\Gamma \vdash \langle C\varrho\beta,\sigma\rangle]\!]_p^*$ terminates on p. We state operational correspondence separately for configurations over variables, expressions, and those over commands; correspondence for arbitrary phrases cannot—and need not—be considered, as configurations are only defined over variables, expressions, and commands, see Section 4.1.

Notation: We use $[\![\Gamma \vdash \sigma]\!]^*$ to denote the encoding of an assignment σ with global variables in $\Gamma \subseteq \mathrm{dom}(\sigma)$. Further, we write $[\![\Gamma \vdash \mathcal{C}]\!]_p^*$ for the encoding of a configuration $\mathcal{C}$ with global variables in Γ. Recall from Chapter 2 that $P \stackrel{\epsilon}{\Longrightarrow} P'$ denotes that either $P' = P$ or $P \stackrel{\tau}{\Longrightarrow} P'$; that is, that there is a sequence of zero or more silent transition.

Lemma 4.5 (Operational Correspondence for Variables and Expressions) *Let $\langle V\varrho\beta,\sigma\rangle$ and $\langle E\varrho\beta,\sigma\rangle$ be configurations over a variable-phrase V, an expression E, or either of both, written $\langle P\varrho\beta,\sigma\rangle$, from CIA-{**await**}. Then,*

- *Transitions of the configurations:*
  1. $\Gamma \vdash \langle V\varrho\beta,\sigma\rangle \stackrel{\surd}{\longrightarrow} \iota$ *implies* $[\![\Gamma \vdash \langle V\varrho\beta,\sigma\rangle]\!]_p^* \stackrel{\bar{p}\langle \mathbf{in}_\iota, \mathbf{out}_\iota\rangle}{\longrightarrow} [\![\Gamma \vdash \sigma]\!]^*$;
  2. $\Gamma \vdash \langle E\varrho\beta,\sigma\rangle \stackrel{\surd}{\longrightarrow} v$ *implies* $[\![\Gamma \vdash \langle E\varrho\beta,\sigma\rangle]\!]_p^* \stackrel{\bar{p}\langle v\rangle}{\Longrightarrow} [\![\Gamma \vdash \sigma]\!]^*$;
  3. $\Gamma \vdash \langle P\varrho\beta,\sigma\rangle \stackrel{\tau}{\longrightarrow} \langle P'\varrho'\beta',\sigma\rangle$ *implies* $[\![\Gamma \vdash \langle P\varrho\beta,\sigma\rangle]\!]_p^* \stackrel{\epsilon}{\Longrightarrow} R' \geq [\![\Gamma \vdash \langle P'\varrho'\beta',\sigma\rangle]\!]_p^*$.
- *Transitions of the encodings:*
  1. $[\![\Gamma \vdash \langle V\varrho\beta,\sigma\rangle]\!]_p^* \stackrel{\bar{p}\langle \mathbf{in}_\iota, \mathbf{out}_\iota\rangle}{\longrightarrow} R$ *implies* $R = [\![\Gamma \vdash \sigma]\!]^*$ *and* $\Gamma \vdash \langle V\varrho\beta,\sigma\rangle \stackrel{\surd}{\Longrightarrow} \iota$;
  2. $[\![\Gamma \vdash \langle E\varrho\beta,\sigma\rangle]\!]_p^* \stackrel{\bar{p}\langle v\rangle}{\longrightarrow} R$ *implies* $R = [\![\Gamma \vdash \sigma]\!]^*$ *and* $\Gamma \vdash \langle E\varrho\beta,\sigma\rangle \stackrel{\surd}{\Longrightarrow} v$;
  3. $[\![\Gamma \vdash \langle P\varrho\beta,\sigma\rangle]\!]_p^* \stackrel{\tau}{\longrightarrow} R$ *implies* $R \geq [\![\Gamma \vdash \langle P'\varrho'\beta',\sigma\rangle]\!]_p^*$ *and* $\Gamma \vdash \langle P\varrho\beta,\sigma\rangle \stackrel{\epsilon}{\Longrightarrow} \langle P'\varrho'\beta',\sigma\rangle$.

Proof: By induction on the structure of variable-phrases and expressions. □

In this operational correspondence, we do not consider that the environment can read from or write to global variables in the π-calculus encoding. This is not necessary, because we use

the results about variables and expressions only together with those for commands, and there variable-accesses from the environment are indeed considered.

We derive operational correspondence for commands based on the results from Lemma 4.5, applying compositional reasoning. Yet, we cannot simply restate the results for silent transitions, because parallel composition of commands within the context of a local store allows different parts of the programs to interfere. Note that in contrast to variables and expressions, commands are capable of modifying the contents of variables.

Lemma 4.6 (Operational Correspondence for Commands) *Let $\langle C\varrho\beta, \sigma\rangle$ be a configuration over a command C from CIA-{**await**}, with global variables $\Gamma \subseteq dom(\sigma)$. Then,*

- *Transitions of the configurations:*

  1. $\Gamma \vdash \langle C\varrho\beta, \sigma\rangle \stackrel{\surd}{\longrightarrow} \sigma'$ *implies* $[\![\Gamma \vdash \langle C\varrho\beta, \sigma\rangle]\!]^*_p \stackrel{\bar{p}}{\Longrightarrow} [\![\Gamma \vdash \sigma']\!]^*$
  2. $\Gamma \vdash \langle C\varrho\beta, \sigma\rangle \stackrel{\overline{\mathbf{in}}_\iota(v)}{\longrightarrow} \langle C\varrho\beta, \sigma\rangle$ *implies* $[\![\Gamma \vdash \langle C\varrho\beta, \sigma\rangle]\!]^*_p \stackrel{\overline{\mathbf{in}}_\iota(v)}{\longrightarrow} [\![\Gamma \vdash \langle C\varrho\beta, \sigma\rangle]\!]^*_p$;
  3. $\Gamma \vdash \langle C\varrho\beta, \sigma\rangle \stackrel{\mathbf{out}_\iota(v)}{\longrightarrow} \langle C\varrho\beta, \sigma'\rangle$ *implies* $[\![\Gamma \vdash \langle C\varrho\beta, \sigma\rangle]\!]^*_p \stackrel{\mathbf{out}_\iota(v)}{\longrightarrow} [\![\Gamma \vdash \langle C\varrho\beta, \sigma'\rangle]\!]^*_p$;
  4. $\Gamma \vdash \langle C\varrho\beta, \sigma\rangle \stackrel{\tau}{\longrightarrow} \langle C'\varrho'\beta', \sigma'\rangle$ *implies* $[\![\Gamma \vdash \langle C\varrho\beta, \sigma\rangle]\!]^*_p \stackrel{\epsilon}{\Longrightarrow} R \geq [\![\Gamma \vdash \langle C'\varrho'\beta', \sigma'\rangle]\!]^*_p$.

- *Transitions of the encodings:*

  1. $[\![\Gamma \vdash \langle C\varrho\beta, \sigma\rangle]\!]^*_p \stackrel{\bar{p}}{\longrightarrow} R$ *implies* $R = [\![\Gamma \vdash \sigma']\!]^*$ *and* $\Gamma \vdash \langle C\varrho\beta, \sigma\rangle \stackrel{\surd}{\Longrightarrow} \sigma'$ *for some* σ';
  2. $[\![\Gamma \vdash \langle C\varrho\beta, \sigma\rangle]\!]^*_p \stackrel{\overline{\mathbf{in}}_\iota(v)}{\longrightarrow} [\![\Gamma \vdash \langle C\varrho\beta, \sigma\rangle]\!]^*_p$ *implies* $\Gamma \vdash \langle C\varrho\beta, \sigma\rangle \stackrel{\overline{\mathbf{in}}_\iota(v)}{\longrightarrow} \langle C\varrho\beta, \sigma\rangle$;
  3. $[\![\Gamma \vdash \langle C\varrho\beta, \sigma\rangle]\!]^*_p \stackrel{\mathbf{out}_\iota(v)}{\longrightarrow} [\![\Gamma \vdash \langle C\varrho\beta, \sigma'\rangle]\!]^*_p$ *implies* $\Gamma \vdash \langle C\varrho\beta, \sigma\rangle \stackrel{\mathbf{out}_\iota(v)}{\longrightarrow} \langle C\varrho\beta, \sigma'\rangle$;
  4. $[\![\Gamma \vdash \langle C\varrho\beta, \sigma\rangle]\!]^*_p \stackrel{\tau}{\longrightarrow} R$ *implies* $R \geq [\![\Gamma \vdash \langle C'\varrho'\beta', \sigma'\rangle]\!]^*_p$ *and* $\Gamma \vdash \langle C\varrho\beta, \sigma\rangle \stackrel{\epsilon}{\Longrightarrow} \langle C'\varrho'\beta', \sigma'\rangle$.

Proof: By induction on the structure of commands, making use of Lemma 4.5. □

Weak operational correspondence So far, in Lemma 4.6, we have compared the small-step behaviour of CIA-{**await**} commands and their translations into the π-calculus. Yet, in the context of observational equivalences, it is as important how the large-step behaviour corresponds. Such a correspondence can easily be obtained from that of the small-step behaviour by composing transitions and reordering administrative steps in an inductive argument.

Lemma 4.7 (Weak Operational Correspondence for Commands) *Let $\langle C\varrho\beta, \sigma\rangle$ be a configuration over a command C from CIA-{**await**}, with global variables $\Gamma \subseteq dom(\sigma)$. Then,*

- *Transitions of the configurations:*

  1. $\Gamma \vdash \langle C\varrho\beta, \sigma\rangle \stackrel{\surd}{\Longrightarrow} \sigma'$ *implies* $[\![\Gamma \vdash \langle C\varrho\beta, \sigma\rangle]\!]^*_p \stackrel{\bar{p}}{\Longrightarrow} [\![\Gamma \vdash \sigma']\!]^*$
  2. $\Gamma \vdash \langle C\varrho\beta, \sigma\rangle \stackrel{\overline{\mathbf{in}}_\iota(v)}{\Longrightarrow} \langle C'\varrho'\beta', \sigma'\rangle$ *implies* $[\![\Gamma \vdash \langle C\varrho\beta, \sigma\rangle]\!]^*_p \stackrel{\overline{\mathbf{in}}_\iota(v)}{\Longrightarrow} R \geq [\![\Gamma \vdash \langle C'\varrho'\beta', \sigma'\rangle]\!]^*_p$;

3. $\Gamma \vdash \langle C\varrho\beta, \sigma\rangle \stackrel{\mathbf{out}_\iota(v)}{\Longrightarrow} \langle C'\varrho'\beta', \sigma'\rangle$ *implies* $[\![\Gamma \vdash \langle C\varrho\beta, \sigma\rangle]\!]^*_p \stackrel{\mathbf{out}_\iota(v)}{\Longrightarrow} R \geq [\![\Gamma \vdash \langle C'\varrho'\beta', \sigma'\rangle]\!]^*_p$;

4. $\Gamma \vdash \langle C\varrho\beta, \sigma\rangle \stackrel{\epsilon}{\Longrightarrow} \langle C'\varrho'\beta', \sigma'\rangle$ *implies* $[\![\Gamma \vdash \langle C\varrho\beta, \sigma\rangle]\!]^*_p \stackrel{\epsilon}{\Longrightarrow} R \geq [\![\Gamma \vdash \langle C'\varrho'\beta', \sigma'\rangle]\!]^*_p$.

- *Transitions of the encodings:*

1. $[\![\Gamma \vdash \langle C\varrho\beta, \sigma\rangle]\!]^*_p \stackrel{\bar{p}}{\Longrightarrow} R$ *implies* $R = [\![\Gamma \vdash \sigma']\!]^*$ *and* $\Gamma \vdash \langle C\varrho\beta, \sigma\rangle \stackrel{\surd}{\Longrightarrow} \sigma'$ *for some* σ';

2. $[\![\Gamma \vdash \langle C\varrho\beta, \sigma\rangle]\!]^*_p \stackrel{\overline{\mathbf{in}}_\iota(v)}{\Longrightarrow} R$ *implies* $R \geq [\![\Gamma \vdash \langle C'\varrho'\beta', \sigma'\rangle]\!]^*_p$ *and* $\Gamma \vdash \langle C\varrho\beta, \sigma\rangle \stackrel{\overline{\mathbf{in}}_\iota(v)}{\Longrightarrow} \langle C'\varrho'\beta', \sigma'\rangle$;

3. $[\![\Gamma \vdash \langle C\varrho\beta, \sigma\rangle]\!]^*_p \stackrel{\mathbf{out}_\iota(v)}{\Longrightarrow} R$ *implies* $R \geq [\![\Gamma \vdash \langle C'\varrho'\beta', \sigma'\rangle]\!]^*_p$ *and* $\Gamma \vdash \langle C\varrho\beta, \sigma\rangle \stackrel{\mathbf{out}_\iota(v)}{\Longrightarrow} \langle C'\varrho'\beta', \sigma'\rangle$;

4. $\Gamma \vdash [\![\langle C\varrho\beta, \sigma\rangle]\!]^*_p \stackrel{\epsilon}{\Longrightarrow} R$ *implies* $R \geq [\![\Gamma \vdash \langle C'\varrho'\beta', \sigma'\rangle]\!]^*_p$ *and* $\Gamma \vdash \langle C\varrho\beta, \sigma\rangle \stackrel{\epsilon}{\Longrightarrow} \langle C'\varrho'\beta', \sigma'\rangle$.

Proof: By induction on the length of the transitions. The argument includes reordering of administrative steps in the expansion-relations and executing steps as captured by the transitions in the π-calculus encoding. □

Soundness Exploiting the congruence-properties of $\approx_\pi$, compositionality of the encoding, and the operational correspondence results from Lemmas 4.6 and 4.7, we can prove that the encoding is sound.

Notation: Let $\approx^-_{oc}$ be the observational congruence on CIA-{**await**} defined analogously to $\approx_{oc}$ on full CIA (see Definitions 4.1 and 4.2), and let $\approx_\pi$ be weak early bisimilarity for the π-calculus, as introduced in Definition 2.4 in Chapter 2. Further, we use $[\![\Gamma \vdash C]\!]^*_p$ to denote the encoding of a closed command C together with an arbitrary assignment to its global variables.

THEOREM 4.8 (FULL ABSTRACTION FOR COMMANDS) *For arbitrary closed commands C_1 and C_2 in CIA-*{**await**}*, with global variables Γ, the following holds:*

- **Soundness:** $[\![\Gamma \vdash C_1]\!]^*_p \approx_\pi [\![\Gamma \vdash C_2]\!]^*_p$ *implies* $\Gamma \vdash C_1 \approx^-_{oc} C_2$;
- **Completeness:** $\Gamma \vdash C_1 \approx^-_{oc} C_2$ *implies* $[\![\Gamma \vdash C_1]\!]^*_p \approx_\pi [\![\Gamma \vdash C_2]\!]^*_p$.

Proof: The proofs of the two directions are similar, and amount to the construction of a configuration-bisimulation from a π-calculus bisimulation, and vice versa. The justification that each of the obtained relations is indeed a bisimulation, is a case-analysis exploiting the results about operational correspondence obtained above.

Soundness: The relation

$$\mathcal{R}_{C1}(\Gamma) \stackrel{\text{def}}{=} \{ (\langle C_1\varrho_1\beta_1, \sigma_1\rangle, \langle C_2\varrho_2\beta_2, \sigma_2\rangle) \mid [\![\Gamma \vdash \langle C_1\varrho_1\beta_1, \sigma_1\rangle]\!]^*_p \approx_\pi [\![\Gamma \vdash \langle C_2\varrho_2\beta_2, \sigma_2\rangle]\!]^*_p \}$$

is a configuration bisimulation according to Definition 4.1. The proof is standard diagram-chasing based on Lemmas 4.6 and 4.7.

Completeness: The relation

$$\mathcal{R}_{C2} \stackrel{\text{def}}{=} \{ ([\![\Gamma \vdash \langle C_1\varrho_1\beta_1, \sigma_1\rangle]\!]^*_p, [\![\Gamma \vdash \langle C_2\varrho_2\beta_2, \sigma_2\rangle]\!]^*_p) \mid \Gamma \vdash \langle C_1\varrho_1\beta_1, \sigma_1\rangle \approx \langle C_2\varrho_2\beta_2, \sigma_2\rangle \}$$
$$\cup \ \mathcal{I}d_{\mathcal{P}}$$

Expressions

$$[\![!V]\!]_p^\ell \stackrel{\text{def}}{=} (\nu\, q)([\![V]\!]_q \,|\, q(i,o).\bar{\ell}.i(x).(\ell.0 \,|\, \bar{p}\langle x\rangle.0))$$

Commands

$$\begin{array}{rcl} [\![V := E]\!]_p^\ell & \stackrel{\text{def}}{=} & (\nu\, q,r)([\![V]\!]_q \,|\, [\![E]\!]_r \,|\, q(i,o).r(x).\bar{\ell}.\bar{o}\langle x\rangle.(\ell.0 \,|\, \bar{p}.0)) \\ [\![\textbf{await}\ B\ \textbf{then}\ C]\!]_p^\ell & \stackrel{\text{def}}{=} & (\nu\, a)(!a.\,(\nu\, q)(\bar{\ell}.[\![B]\!]_q \,|\, q(x).\,([x = tt]\,(\nu\, r)([\![C]\!]_r \,|\, r.(\ell.0 \,|\, \bar{p}.0)) \,| \\ & & \qquad [x = f\!f]\,(\ell.0 \,|\, \bar{a}.0))) \,|\, \bar{a}.0) \end{array}$$

Table 4.7: **Encoding Full CIA.** We only state those defining clauses differing from Table 4.6.

is a weak bisimulation up to expansion, where $\mathcal{I}d_{\mathcal{P}}$ is the identity-relation on processes. Again, the proof is standard diagram-chasing based on Lemmas 4.6 and 4.7. □

From this result, we immediately obtain soundness of the original encoding for arbitrary phrases from CIA-{**await**}.

THEOREM 4.9 (SOUNDNESS) $[\![P_1]\!]_p \approx_\pi [\![P_2]\!]_p$ *implies* $\Gamma \vdash P_1 \approx_{oc}^- P_2$ *for arbitrary CIA-*{**await**} *phrases* P_1 *and* P_2.

Proof: The result is a corollary of Theorem 4.8, following by an application of the usual congruence properties of $\approx_\pi$. That is, if $[\![P_1]\!]_p \approx_\pi [\![P_2]\!]_p$, and hence $[\![P_1]\!]_p^* \approx_\pi [\![P_2]\!]_p^*$, then for all closing contexts $\Gamma' \vdash Con[.]$, it holds that $[\![\Gamma' \vdash Con[P_1]]\!]_p^* \approx_\pi [\![\Gamma' \vdash Con[P_2]]\!]_p^*$. By Theorem 4.8, we obtain for all closing contexts $Con[.]$, that $\Gamma' \vdash Con[P_1] \approx_{oc}^- Con[P_2]$. Hence $\Gamma \vdash P_1 \approx_{oc}^- P_2$, by Definition 4.2. □

The converse (completeness) holds in the case of closed commands, but does not extend to arbitrary phrases, as we shall discuss in Section 4.4. To give an intuition, one reason is that when giving the environment direct access to local variables by submitting a procedure-call, for instance, the environment retains this access permanently, even after it has signalled termination of the procedure. Some of these problems can be attacked by using type-systems, but it is unclear whether completeness can be obtained with standard type-systems.

4.2.2 Encoding Full CIA

Although during the execution of an **await**-statement, the whole context is stopped, it is sufficient simply to restrict accesses to variables in the π-calculus translation. The reason ist that any other step does not change the semantics of the encoding with respect to weak bisimilarity.

Encoding the lock We encode command-configurations $\langle C\varrho\beta, \sigma\rangle$ as describe above, only that we now add a lock $\ell.0$ in parallel. A simple possibility to hide the lock from the environment is to restrict the whole encoding with respect to it, that is, to give a translation $(\nu\ell)(\ell.0 \,|\, [\![\Gamma \vdash \langle C\varrho\beta, \sigma\rangle]\!]_p^\ell)$. Such an encoding will work well in those cases where it does not matter whether the environment interferes during the execution of an **await**-statement. A major problem of this simple solution, however, is that this is not always the case. Note

that, according to our operational semantics, the environment is intended to be blocked by an **await**-statement as well. This implies that even certain obvious correspondences cannot be proved via a translation into the π-calculus, as for instance,

$$\iota : \mathbf{var}[int] \;\vdash\; \mathbf{await}\ tt\ \mathbf{then}\ (\iota := \iota + 1; \iota := \iota + 1) \;\approx_{oc}\; \mathbf{await}\ tt\ \mathbf{then}\ (\iota := \iota + 2).$$

The reason is that the I/O-typing we have applied so far does not forbid the environment access to $\mathbf{Reg}_\iota$. It suffices for the environment to read from $\mathbf{Reg}_\iota$ as soon as the first assignment has taken place in the first command; the second command cannot change the value of $\mathbf{Reg}_\iota$ accordingly.

Linearity We thus have to require from the environment to take the lock before accessing a variable, and to release it afterwards. This means that we have to make the lock public, hence encode configurations via $\ell.0 \,|\, [\![\Gamma \vdash \langle C\varrho\beta, \sigma\rangle]\!]^{\ell}_{p}$, and make sure that the lock is taken by the environment only for accessing global variables, after which it is to be released again. This can be encoded in the obligations for weak bisimilarity (see Definition 2.4) as follows (we only state the changes for the first one):

- if $P \xrightarrow{\ell} \xrightarrow{\overline{\mathbf{in}}_\iota\langle v\rangle} P'$, then there exists Q' such that $Q \xRightarrow{\ell} \xRightarrow{\overline{\mathbf{in}}_\iota\langle v\rangle} Q'$ and $\ell.0 \,|\, P' \,\mathcal{R}\, \ell.0 \,|\, Q'$;
- if $P \xrightarrow{\ell} \xrightarrow{\mathbf{out}_\iota(v)} P'$, then there exists Q' such that $Q \xRightarrow{\ell} \xRightarrow{\overline{\mathbf{in}}_\iota(v)} Q'$ and $\ell.0 \,|\, P' \,\mathcal{R}\, \ell.0 \,|\, Q'$;
- if $P \xrightarrow{\tau} P'$, then there exists Q' such that $Q \xRightarrow{\epsilon} Q'$ and $P' \,\mathcal{R}\, Q'$.

Like this, it is guaranteed that the lock is acquired by the environment before reading from or writing to the global variables, and that the lock is released again afterwards. Note that the above is an encoding into the definition of weak bisimilarity, of linear types as proposed by Kobayashi, Pierce, and Turner [KPT99]. Using this modification, the above example can indeed be validated.

An auxiliary encoding Like in the previous section, we employ an auxiliary encoding $[\![P]\!]^{\ell*}_p$ of phrases P such that $[\![P]\!]_p \geq [\![P]\!]^{\ell*}_p$; it is obtained by performing all administrative steps. By compositionality of expansion, this again extends to whole configurations, that is, $[\![\Gamma \vdash \langle P\varrho\beta, \sigma\rangle]\!]_p \geq [\![\Gamma \vdash \langle P\varrho\beta, \sigma\rangle]\!]^{\ell*}_p$. Also like above, we use $[\![\Gamma \vdash \sigma]\!]^{\ell*}$ as the encoding of an assignment σ with global variables in $\Gamma \subseteq \mathrm{dom}(\sigma)$.

Operational correspondence When deriving operational correspondence for locked configurations, we can make use of the fact that the execution of an **await**-statement is fully deterministic, and hence can be dealt with bisimulations up-to expansion, both for CIA and the encoding.

Note that already in the lemmas about operational correspondence, we do already apply locks so that the encodings can access them. Yet, we do not yet force the environment to behave according to a regiment of linear typing.

LEMMA 4.10 (OPERATIONAL CORRESPONDENCE FOR VARIABLES AND EXPRESSIONS) *Let* $\langle V\varrho\beta, \sigma\rangle$ *and* $\langle E\varrho\beta, \sigma\rangle$ *be configurations over a variable-phrase* V*, an expression* E*, or either of both, written* $\langle P\varrho\beta, \sigma\rangle$*, from full CIA. Then,*

- *Transitions of the configurations:*

1. $\Gamma \vdash \langle V\varrho\beta, \sigma\rangle \stackrel{\surd}{\longrightarrow} \iota$ *implies* $[\![\Gamma \vdash \langle V\varrho\beta, \sigma\rangle]\!]_p^{\ell*} \stackrel{\bar{p}\langle \mathbf{in}_\iota, \mathbf{out}_\iota\rangle}{\longrightarrow} [\![\Gamma \vdash \sigma]\!]^{\ell*}$;

2. $\Gamma \vdash \langle E\varrho\beta, \sigma\rangle \stackrel{\surd}{\longrightarrow} v$ *implies* $\ell.0 \,|\, [\![\Gamma \vdash \langle E\varrho\beta, \sigma\rangle]\!]_p^{\ell*} \stackrel{\bar{p}\langle v\rangle}{\Longrightarrow} \ell.0 \,|\, [\![\Gamma \vdash \sigma]\!]^{\ell*}$;

3. $\Gamma \vdash \langle P\varrho\beta, \sigma\rangle \stackrel{\tau}{\longrightarrow} \langle P'\varrho'\beta', \sigma\rangle$ *implies* $\ell.0 \,|\, [\![\Gamma \vdash \langle P\varrho\beta, \sigma\rangle]\!]_p^{\ell*} \stackrel{\epsilon}{\Longrightarrow} R' \geq \ell.0 \,|\, [\![\Gamma \vdash \langle P'\varrho'\beta', \sigma\rangle]\!]_p^{\ell*}$.

- *Transitions of the encodings:*

1. $[\![\Gamma \vdash \langle V\varrho\beta, \sigma\rangle]\!]_p^{\ell*} \stackrel{\bar{p}\langle \mathbf{in}_\iota, \mathbf{out}_\iota\rangle}{\longrightarrow} R$ *implies* $R = [\![\Gamma \vdash \sigma]\!]^{\ell*}$ *and* $\Gamma \vdash \langle V\varrho\beta, \sigma\rangle \stackrel{\surd}{\Longrightarrow} \iota$;

2. $[\![\Gamma \vdash \langle E\varrho\beta, \sigma\rangle]\!]_p^{\ell*} \stackrel{\bar{p}\langle v\rangle}{\longrightarrow} R$ *implies* $R = [\![\Gamma \vdash \sigma]\!]^{\ell*}$ *and* $\Gamma \vdash \langle E\varrho\beta, \sigma\rangle \stackrel{\surd}{\Longrightarrow} v$;

3. $\ell.0 \,|\, [\![\Gamma \vdash \langle P\varrho\beta, \sigma\rangle]\!]_p^{\ell*} \stackrel{\tau}{\longrightarrow} R$ *implies* $R \geq \ell.0 \,|\, [\![\Gamma \vdash \langle P'\varrho'\beta', \sigma\rangle]\!]_p^{\ell*}$ *and* $\Gamma \vdash \langle P\varrho\beta, \sigma\rangle \stackrel{\epsilon}{\Longrightarrow} \langle P'\varrho'\beta', \sigma\rangle$.

Proof: By induction on the structure of variable-phrases and expressions. □

LEMMA 4.11 (OPERATIONAL CORRESPONDENCE FOR COMMANDS) *Let $\langle C\varrho\beta, \sigma\rangle$ be a configuration over a command C from full CIA, with global variables $\Gamma \subseteq dom(\sigma)$. Then,*

- *Transitions of the configurations:*

1. $\Gamma \vdash \langle C\varrho\beta, \sigma\rangle \stackrel{\surd}{\longrightarrow} \sigma'$ *implies* $\ell.0 \,|\, [\![\Gamma \vdash \langle C\varrho\beta, \sigma\rangle]\!]_p^{\ell*} \stackrel{\bar{p}}{\Longrightarrow} \ell.0 \,|\, [\![\Gamma \vdash \sigma']\!]^{\ell*}$

2. $\Gamma \vdash \langle C\varrho\beta, \sigma\rangle \stackrel{\overline{\mathbf{in}}_\iota(v)}{\longrightarrow} \langle C\varrho\beta, \sigma\rangle$ *implies* $[\![\Gamma \vdash \langle C\varrho\beta, \sigma\rangle]\!]_p^{\ell*} \stackrel{\overline{\mathbf{in}}_\iota(v)}{\longrightarrow} [\![\Gamma \vdash \langle C\varrho\beta, \sigma\rangle]\!]_p^{\ell*}$;

3. $\Gamma \vdash \langle C\varrho\beta, \sigma\rangle \stackrel{\mathbf{out}_\iota(v)}{\longrightarrow} \langle C\varrho\beta, \sigma'\rangle$ *implies* $[\![\Gamma \vdash \langle C\varrho\beta, \sigma\rangle]\!]_p^{\ell*} \stackrel{\mathbf{out}_\iota(v)}{\longrightarrow} [\![\Gamma \vdash \langle C\varrho\beta, \sigma'\rangle]\!]_p^{\ell*}$;

4. $\Gamma \vdash \langle C\varrho\beta, \sigma\rangle \stackrel{\tau}{\longrightarrow} \langle C'\varrho'\beta', \sigma'\rangle$ *implies* $\ell.0 \,|\, [\![\Gamma \vdash \langle C\varrho\beta, \sigma\rangle]\!]_p^{\ell*} \stackrel{\epsilon}{\Longrightarrow} R \geq \ell.0 \,|\, [\![\Gamma \vdash \langle C'\varrho'\beta', \sigma'\rangle]\!]_p^{\ell*}$;

5. $\Gamma \vdash \langle C\varrho\beta, \sigma\rangle \stackrel{\tau}{\longrightarrow} \langle C'\varrho\beta, \sigma\rangle_\ell$ *implies* $\langle C'\varrho\beta, \sigma\rangle_\ell \stackrel{\epsilon}{\Longrightarrow} \langle C''\varrho\beta, \sigma'\rangle$ *and* $\langle C'\varrho\beta, \sigma\rangle_\ell \approx \langle C''\varrho\beta, \sigma'\rangle$ *and* $\ell.0 \,|\, [\![\Gamma \vdash \langle C\varrho\beta, \sigma\rangle]\!]_p^{\ell*} \stackrel{\epsilon}{\Longrightarrow} R \geq \ell.0 \,|\, [\![\Gamma \vdash \langle C''\varrho\beta, \sigma'\rangle]\!]_p^{\ell*}$.

- *Transitions of the encodings:*

1. $[\![\Gamma \vdash \langle C\varrho\beta, \sigma\rangle]\!]_p^{\ell*} \stackrel{\bar{p}}{\longrightarrow} R$ *implies* $R = [\![\Gamma \vdash \sigma']\!]^{\ell*}$ *and* $\Gamma \vdash \langle C\varrho\beta, \sigma\rangle \stackrel{\surd}{\Longrightarrow} \sigma'$ *for some* σ';

2. $[\![\Gamma \vdash \langle C\varrho\beta, \sigma\rangle]\!]_p^{\ell*} \stackrel{\overline{\mathbf{in}}_\iota(v)}{\longrightarrow} [\![\Gamma \vdash \langle C\varrho\beta, \sigma\rangle]\!]_p^{\ell*}$ *implies* $\Gamma \vdash \langle C\varrho\beta, \sigma\rangle \stackrel{\overline{\mathbf{in}}_\iota(v)}{\longrightarrow} \langle C\varrho\beta, \sigma\rangle$;

3. $[\![\Gamma \vdash \langle C\varrho\beta, \sigma\rangle]\!]_p^{\ell*} \stackrel{\mathbf{out}_\iota(v)}{\longrightarrow} [\![\Gamma \vdash \langle C\varrho\beta, \sigma'\rangle]\!]_p^{\ell*}$ *implies* $\Gamma \vdash \langle C\varrho\beta, \sigma\rangle \stackrel{\mathbf{out}_\iota(v)}{\longrightarrow} \langle C\varrho\beta, \sigma'\rangle$;

4. $\ell.0 \,|\, [\![\Gamma \vdash \langle C\varrho\beta, \sigma\rangle]\!]_p^{\ell*} \stackrel{\tau}{\longrightarrow} R$ *implies* $R \geq \ell.0 \,|\, [\![\Gamma \vdash \langle C'\varrho'\beta', \sigma'\rangle]\!]_p^{\ell*}$ *and* $\Gamma \vdash \langle C\varrho\beta, \sigma\rangle \stackrel{\epsilon}{\Longrightarrow} \langle C'\varrho'\beta', \sigma'\rangle$.

Proof: By induction on the structure of commands, making use of Lemma 4.10. □

Note that the context of the encoding can principally access the variables without taking the lock. We omit the respective branches of the transition-systems when examining operational correspondence, because they are eliminated by the linearity-constraints imposed by configuration-bisimulation. As a result, we do not even have to add the locks when considering the visible transitions of the encodings. In the results for weak operational correspondence below, we have to add the locks nevertheless, because they can be taken and released in the silent transitions happening before and after the inputs and outputs.

LEMMA 4.12 (WEAK OPERATIONAL CORRESPONDENCE FOR COMMANDS) *Let $\langle C\varrho\beta, \sigma\rangle$ be a configuration over a command C from full CIA, with global variables $\Gamma \subseteq dom(\sigma)$. Then,*

- *Transitions of the configurations:*

  1. $\Gamma \vdash \langle C\varrho\beta, \sigma\rangle \stackrel{\surd}{\Longrightarrow} \sigma'$ *implies* $\ell.0 \,|\, [\![\Gamma \vdash \langle C\varrho\beta, \sigma\rangle]\!]_p^{\ell*} \stackrel{\bar{p}}{\Longrightarrow} \ell.0 \,|\, [\![\Gamma \vdash \sigma']\!]^{\ell*}$
  2. $\Gamma \vdash \langle C\varrho\beta, \sigma\rangle \stackrel{\overline{\mathbf{in}}_\iota(v)}{\Longrightarrow} \langle C'\varrho'\beta', \sigma'\rangle$ *implies* $\ell.0 \,|\, [\![\Gamma \vdash \langle C\varrho\beta, \sigma\rangle]\!]_p^{\ell*} \stackrel{\overline{\mathbf{in}}_\iota(v)}{\Longrightarrow} R \geq \ell.0 \,|\, [\![\Gamma \vdash \langle C'\varrho'\beta', \sigma'\rangle]\!]_p^{\ell*}$;
  3. $\Gamma \vdash \langle C\varrho\beta, \sigma\rangle \stackrel{\mathbf{out}_\iota(v)}{\Longrightarrow} \langle C'\varrho'\beta', \sigma'\rangle$ *implies* $\ell.0 \,|\, [\![\Gamma \vdash \langle C\varrho\beta, \sigma\rangle]\!]_p^{\ell*} \stackrel{\mathbf{out}_\iota(v)}{\Longrightarrow} R \geq \ell.0 \,|\, [\![\Gamma \vdash \langle C'\varrho'\beta', \sigma'\rangle]\!]_p^{\ell*}$;
  4. $\Gamma \vdash \langle C\varrho\beta, \sigma\rangle \stackrel{\epsilon}{\Longrightarrow} \langle C'\varrho'\beta', \sigma'\rangle$ *implies* $\ell.0 \,|\, [\![\Gamma \vdash \langle C\varrho\beta, \sigma\rangle]\!]_p^{\ell*} \stackrel{\epsilon}{\Longrightarrow} R \geq \ell.0 \,|\, [\![\Gamma \vdash \langle C'\varrho'\beta', \sigma'\rangle]\!]_p^{\ell*}$.
  5. $\Gamma \vdash \langle C\varrho\beta, \sigma\rangle \stackrel{\epsilon}{\Longrightarrow} \langle C'\varrho\beta, \sigma\rangle_\ell$ *implies* $\langle C'\varrho\beta, \sigma\rangle_\ell \stackrel{\epsilon}{\Longrightarrow} \langle C''\varrho\beta, \sigma'\rangle$ *and* $\langle C'\varrho\beta, \sigma\rangle_\ell \approx \langle C''\varrho\beta, \sigma'\rangle$ *and* $\ell.0 \,|\, [\![\Gamma \vdash \langle C\varrho\beta, \sigma\rangle]\!]_p^{\ell*} \stackrel{\epsilon}{\Longrightarrow} R \geq \ell.0 \,|\, [\![\Gamma \vdash \langle C''\varrho\beta, \sigma'\rangle]\!]_p^{\ell*}$.

- *Transitions of the encodings:*

  1. $\ell.0 \,|\, [\![\Gamma \vdash \langle C\varrho\beta, \sigma\rangle]\!]_p^{\ell*} \stackrel{\bar{p}}{\Longrightarrow} R$ *implies* $R = \ell.0 \,|\, [\![\Gamma \vdash \sigma']\!]^{\ell*}$ *and* $\Gamma \vdash \langle C\varrho\beta, \sigma\rangle \stackrel{\surd}{\Longrightarrow} \sigma'$ *for some* σ';
  2. $\ell.0 \,|\, [\![\Gamma \vdash \langle C\varrho\beta, \sigma\rangle]\!]_p^{\ell*} \stackrel{\overline{\mathbf{in}}_\iota(v)}{\Longrightarrow} R$ *implies* $R \geq \ell.0 \,|\, [\![\Gamma \vdash \langle C'\varrho'\beta', \sigma'\rangle]\!]_p^{\ell*}$ *and* $\Gamma \vdash \langle C\varrho\beta, \sigma\rangle \stackrel{\overline{\mathbf{in}}_\iota(v)}{\Longrightarrow} \langle C'\varrho'\beta', \sigma'\rangle$;
  3. $\ell.0 \,|\, [\![\Gamma \vdash \langle C\varrho\beta, \sigma\rangle]\!]_p^{\ell*} \stackrel{\mathbf{out}_\iota(v)}{\Longrightarrow} R$ *implies* $R \geq \ell.0 \,|\, [\![\Gamma \vdash \langle C'\varrho'\beta', \sigma'\rangle]\!]_p^{\ell*}$ *and* $\Gamma \vdash \langle C\varrho\beta, \sigma\rangle \stackrel{\mathbf{out}_\iota(v)}{\Longrightarrow} \langle C'\varrho'\beta', \sigma'\rangle$;
  4. $\ell.0 \,|\, [\![\Gamma \vdash \langle C\varrho\beta, \sigma\rangle]\!]_p^{\ell*} \stackrel{\epsilon}{\Longrightarrow} R$ *implies* $R \geq \ell.0 \,|\, [\![\Gamma \vdash \langle C'\varrho'\beta', \sigma'\rangle]\!]_p^{\ell*}$ *and* $\Gamma \vdash \langle C\varrho\beta, \sigma\rangle \stackrel{\epsilon}{\Longrightarrow} \langle C'\varrho'\beta', \sigma'\rangle$.

Proof: By induction on the length of the transitions. The argument includes reordering of administrative steps in the expansion relations and executing steps as captured by the transitions in the π-calculus encoding. □

Up-to techniques in the proof of Theorem 4.13 (full abstraction for commands) below allow us to abstract from locked configurations. On the level of CIA, we employ the results from Corollary 4.4, on that of the π-calculus we use expansion.

Soundness Exploiting the congruence-properties of $\approx_\pi$, the compositionality of the encoding, and the operational correspondence results, we can prove that the encoding is sound.

Let $\approx_{oc}$ be the observational congruence on full CIA, as given in Definitions 4.1 and 4.2, and let $\approx_\pi$ be weak early bisimilarity with linear types for the π-calculus, as refined from in Definition 2.4 in Chapter 2 above.

THEOREM 4.13 (FULL ABSTRACTION FOR COMMANDS) *For arbitrary closed commands C_1 and C_2 in full CIA, with global variables Γ, the following holds:*

- **Soundness:** $[\![\Gamma \vdash C_1]\!]_p^{\ell*} \approx_\pi [\![\Gamma \vdash C_2]\!]_p^{\ell*}$ *implies* $\Gamma \vdash C_1 \approx_{oc}^{-} C_2$*;*
- **Completeness:** $\Gamma \vdash C_1 \approx_{oc}^{-} C_2$ *implies* $[\![\Gamma \vdash C_1]\!]_p^{\ell*} \approx_\pi [\![\Gamma \vdash C_2]\!]_p^{\ell*}$*.*

Proof: The proofs of the two directions are similar, and amount to the construction of a configuration bisimulation from a π-calculus bisimulation, and vice versa. The justification that each of the obtained relations is indeed a bisimulation, is a case-analysis exploiting the results about operational correspondence obtained above.

Soundness: The relation

$$\begin{aligned} \mathcal{R}_{C1}(\Gamma) \stackrel{\text{def}}{=} \{ (\langle C_1\varrho_1\beta_1, \sigma_1\rangle, \langle C_2\varrho_2\beta_2, \sigma_2\rangle) \mid \\ \ell.0 \mid [\![\Gamma \vdash \langle C_1\varrho_1\beta_1, \sigma_1\rangle]\!]_p^{\ell*} \approx_\pi \ell.0 \mid [\![\Gamma \vdash \langle C_2\varrho_2\beta_2, \sigma_2\rangle]\!]_p^{\ell*} \} \end{aligned}$$

is a configuration bisimulation up to $\approx$ according to Definition 4.1 when taking linear typing of the lock into accout. The proof is standard diagram-chasing based on Lemmas 4.11 and 4.12.

Completeness: The relation

$$\begin{aligned} \mathcal{R}_{C2} \stackrel{\text{def}}{=} & \{ (\ell.0 \mid [\![\Gamma \vdash \langle C_1\varrho_1\beta_1, \sigma_1\rangle]\!]_p^{\ell*}, \ell.0 \mid [\![\Gamma \vdash \langle C_2\varrho_2\beta_2, \sigma_2\rangle]\!]_p^{\ell*}) \mid \\ & \Gamma \vdash \langle C_1\varrho_1\beta_1, \sigma_1\rangle \approx \langle C_2\varrho_2\beta_2, \sigma_2\rangle \} \\ \cup & \ \mathcal{I}d_{\mathcal{P}} \end{aligned}$$

is a weak bisimulation up to expansion, with linear typing for the lock; again, $\mathcal{I}d_{\mathcal{P}}$ is the identity-relation on processes. Again, the proof is standard diagram-chasing based on Lemmas 4.11 and 4.12. □

From this result, we immediately obtain soundness for arbitrary phrases. We use $\approx_\pi$ to denote weak bisimilarity for the π-calculus, and $\approx_\pi^l$ to stand for weak bisimilarity with a linear typing.

THEOREM 4.14 (SOUNDNESS) $\ell.0 \mid [\![P_1]\!]_p^\ell \approx_\pi^l \ell.0 \mid [\![P_2]\!]_p^\ell$ *and* $(\nu\ell)(\ell.0 \mid [\![P_1]\!]_p^\ell) \approx_\pi (\nu\ell)(\ell.0 \mid [\![P_2]\!]_p^\ell)$ *imply* $\Gamma \vdash P_1 \approx_{oc} P_2$ *for arbitrary CIA-phrases P_1 and P_2.*

Proof: The result is a corollary of Theorem 4.13, following by an application of the usual congruence properties of $\approx_\pi^l$. Clearly, $(\nu\ell)(\ell.0 \mid [\![P_1]\!]_p^\ell) \approx_\pi (\nu\ell)(\ell.0 \mid [\![P_2]\!]_p^\ell)$ implies $\ell.0 \mid [\![P_1]\!]_p^\ell \approx_\pi^l \ell.0 \mid [\![P_2]\!]_p^\ell$. Further, if $\ell.0 \mid [\![P_1]\!]_p^\ell \approx_\pi^l \ell.0 \mid [\![P_2]\!]_p^\ell$, and hence $\ell.0 \mid [\![P_1]\!]_p^{\ell*} \approx_\pi \ell.0 \mid [\![P_2]\!]_p^{\ell*}$, then for all closing contexts $\Gamma' \vdash Con[.]$, it holds that $\ell.0 \mid [\![\Gamma' \vdash Con[P_1]]\!]_p^{\ell*} \approx_\pi \ell.0 \mid [\![\Gamma' \vdash Con[P_2]]\!]_p^{\ell*}$. By Theorem 4.13, we obtain for all closing contexts $Con[.]$, that $\Gamma' \vdash Con[P_1] \approx_{oc} Con[P_2]$. Hence $\Gamma \vdash P_1 \approx_{oc} P_2$, by Definition 4.2. □

Relating locked and unlocked encodings The locked encoding $\ell.0 \mid [\![P]\!]_p^\ell$ of a phrase P not containing an **await**-statement, is equivalent to its unlocked encoding $\ell.0 \mid [\![P]\!]_p$, when applying the linear typing-regiment for the lock. Further, it holds for arbitrary phrases P and P' without **await** that $[\![P]\!]_p \approx_\pi [\![P']\!]_p$ implies $\ell.0 \mid [\![P]\!]_p^\ell \approx_\pi \ell.0 \mid [\![P']\!]_p^\ell$. Therefore, we can apply the simpler encoding for CIA-{**await**} whenever possible, in order to derive corresponding results for full CIA.

4.3 Laws and Examples

We illustrate how to use π-calculus semantics by validating laws as well as concrete examples, of CIA. We prove most of the laws in a purely algebraic way, using some simple well-known π-calculus laws. In particular, we consider benchmark laws and examples from [MS88, Bro96, MT91, MT92], demonstrating that the π-calculus semantics yields simple proofs of these well-known equalities. In Section 4.3.1, we then show by a more complex example how to tackle procedures of higher order. In that example, we apply the proof-method of exhibiting a bisimulation, as introduced in Section 2.2.2 and applied in further case-studies in Chapter 5.

• Basic properties of CIA operators, such as associativity of sequential composition, or associativity and commutativity of parallel composition, are straightforward consequences of analogous π-calculus laws (like associativity and commutativity of parallel composition in the π-calculus).

• Suppose that ι does not occur free in P', and consider the following laws:

$$\begin{array}{ll}
(\text{L1}) & \mathbf{new}\ [\tau]\iota := v\ \mathbf{in}\ P' \ = \ P' \\
(\text{L2}) & \mathbf{new}\ [\tau]\iota := v\ \mathbf{in}\ (P; P') \ = \ (\mathbf{new}\ [\tau]\,\iota := v\ \mathbf{in}\ P); P' \\
(\text{L3}) & \mathbf{new}\ [\tau]\iota := v\ \mathbf{in}\ (P'; P) \ = \ P'; (\mathbf{new}\ [\tau]\iota := v\ \mathbf{in}\ P) \\
(\text{L4}) & \mathbf{new}\ [\tau]\iota := v\ \mathbf{in}\ (P \parallel P') \ = \ (\mathbf{new}\ [\tau]\iota := v\ \mathbf{in}\ P) \parallel P'.
\end{array}$$

The π-calculus proofs of these laws are all similar, and purely algebraic. As an example, we present the proof of L2; recall from Section 4.2 that $fn_\iota \stackrel{\text{def}}{=} \{\mathbf{in}_\iota, \mathbf{out}_\iota\}$:

$$\begin{array}{rll}
[\![\mathbf{new}\ [\tau]\iota{:=}v\ \mathbf{in}\ (P; P')]\!]_p & \approx_\pi (\nu\, fn_\iota)(\mathbf{Reg}_\iota[v] \mid (\nu\, q)([\![P]\!]_q \mid q.[\![P']\!]_p)) & (1) \\
& \approx_\pi (\nu\, q)((\nu\, fn_\iota)(\mathbf{Reg}_\iota[v] \mid ([\![P]\!]_q \mid q.[\![P']\!]_p))) & (2) \\
& \approx_\pi (\nu\, q)((\nu\, fn_\iota)(\mathbf{Reg}_\iota[v] \mid [\![P]\!]_q) \mid q.[\![P']\!]_p) & (3) \\
& \approx_\pi [\![(\mathbf{new}\ [\tau]\iota := v\ \mathbf{in}\ P); P')]\!]_p. &
\end{array}$$

Line (1) contains the encoding with v already written to $\mathbf{Reg}_\iota$; in Section 4.2.1, we have shown that this process is bisimilar to the original encoding. In line (2), the restriction on q is moved to an outer level; and in line (3), the restriction on fn_ι is finally removed from $[\![P']\!]_p$.

• A general law for call-by-name languages states that global procedures with read-capacity only cannot change the content of local variables, which allows for an encoding of call-by-value procedures $P(E)$ if the argument is an expression: declare a fresh local variable ι, and evaluate E into it; then call $P(!\iota)$. The following instance of the law,

$$\mathbf{new}\ [int]\iota := 1\ \mathbf{in}\ P(!\iota) \ = \ P(1)$$

(where P is a free identifier of appropriate type) is proved algebraically as follows:

$$
\begin{array}{lll}
\multicolumn{3}{l}{[\![\mathbf{new}\ [int]\iota := 1\ \mathbf{in}\ P(!\iota)]\!]_p} \\
\approx_\pi & (\nu\, fn_\iota)(\mathbf{Reg}_\iota[1] \mid (\nu\, q)(\overline{x}_P\langle q\rangle.0 \mid q(v).(\nu\, x)(\overline{v}\langle x,p\rangle.!x(r).\mathbf{in}_\iota(z).\overline{r}\langle z\rangle.0))) & (1) \\
\approx_\pi & (\nu\, q)(\overline{x}_P\langle q\rangle.0 \mid q(v).(\nu\, x)(\overline{v}\langle x,p\rangle.(\nu\, fn_\iota)(\mathbf{Reg}_\iota[1] \mid !x(r).\mathbf{in}_\iota(z).\overline{r}\langle z\rangle.0))) & (2) \\
\approx_\pi & (\nu\, q)(\overline{x}_P\langle q\rangle.0 \mid q(v).(\nu\, x)(\overline{v}\langle x,p\rangle.!x(r).\overline{r}\langle 1\rangle.0)) & (3) \\
= & [\![P(1)]\!]_p. &
\end{array}
$$

Line (1) again contains the encoding with the declaration already performed (compare the example above and Section 4.2.1). In lines (2) and (3), $\mathbf{Reg}_\iota$ is eliminated owing to $\mathbf{in}_\iota$ being the only name in fn_ι to be used: in line (2), $\mathbf{Reg}_\iota$ is moved into the only sub-process accessing it, and the binder $(\nu\, fn_\iota)$ is moved inside the process accordingly; in line (3), $\mathbf{Reg}_\iota$ is finally eliminated as, in the absence of $\mathbf{out}_\iota$, reading and returning a value is equivalent to just returning the value itself.

• Another important law for IA and CIA is that local variables that are modified but never used afterwards, are superfluous and can be omitted. Suppose again that P is a free identifier of appropriate type. Then, calling it with a command modifying a local variable ι amounts to calling it with the **skip**-argument, provided ι is not read from afterwards. Proving the law,

$$\mathbf{new}\ [int]\iota := 0\ \mathbf{in}\ P(\iota :=!\iota + 1) \quad = \quad P(\mathbf{skip})$$

essentially amounts to showing for arbitrary non-negative integer values v that,

$$(\nu\, fn_\iota)(\mathbf{Reg}_\iota[v] \mid !x(r).[\![\iota :=!\iota + 1]\!]_r) \quad \approx_\pi \quad !x(r).[\![\mathbf{skip}]\!]_r,$$

where x denotes the formal parameter of the free identifier P. Observe that x is a free name accessible by the observer, because P is a free identifier. We sketch the proof that the two processes are bisimilar. The observer can invoke the argument via x, forking copies of $[\![\iota :=!\iota + 1]\!]_q$ and $[\![\mathbf{skip}]\!]_q$ respectively and providing them with a fresh name q on which to signal termination:

$$
\begin{array}{rl}
(\nu\, fn_\iota)(\mathbf{Reg}_\iota[v] \mid !x(r).[\![\iota:=!\iota + 1]\!]_r) \xrightarrow{x\langle q\rangle} & (\nu\, fn_\iota)(\mathbf{Reg}_\iota[v] \mid !x(r).[\![\iota:=!\iota + 1]\!]_r | [\![\iota:=!\iota + 1]\!]_q) \\
!x(r).[\![\mathbf{skip}]\!]_r \xrightarrow{x\langle q\rangle} & !x(r).[\![\mathbf{skip}]\!]_r \mid [\![\mathbf{skip}]\!]_q,
\end{array}
$$

Assuming that no further procedure-calls occur while the present call is being handled, we have:

$$
\begin{array}{lll}
\multicolumn{3}{l}{(\nu\, fn_\iota)(\mathbf{Reg}_\iota[v] \mid !x(r).[\![\iota :=!\iota + 1]\!]_r \mid [\![\iota :=!\iota + 1]\!]_q)} \\
\stackrel{\epsilon}{\Longrightarrow} & (\nu\, fn_\iota)(\mathbf{Reg}_\iota[v+1] \mid !x(r).[\![\iota :=!\iota + 1]\!]_r \mid [\![\mathbf{skip}]\!]_q) & (1) \\
\xrightarrow{\overline{q}} & (\nu\, fn_\iota)(\mathbf{Reg}_\iota[v+1] \mid !x(r).[\![\iota :=!\iota + 1]\!]_r) & (2) \\
\multicolumn{2}{l}{!x(r).[\![\mathbf{skip}]\!]_r \mid [\![\mathbf{skip}]\!]_q \xrightarrow{\overline{q}} !x(r).[\![\mathbf{skip}]\!]_r.} & (3)
\end{array}
$$

Line (1) represents the internal actions performed when incrementing the value stored in $\mathbf{Reg}_\iota$, and lines (2) and (3) show the signals of termination of the execution of the respective arguments, yielding processes according the same scheme as before (only with a different value).

The general case, in which the argument may be invoked arbitrarily often before the termination of a previous call, can be proved analogously. In this case, more pending processes have

to be added in parallel; they represent both copies of the argument and components occurring whenever one of these copies has read a value from ι but not yet written back the result of its incrementation.

• A simple π-calculus bisimulation-relation can be used to prove that iteration is expressible via recursion, that is, if x is not free in B and C then,

$$\textbf{while } B \textbf{ do } C \;=\; \textbf{rec}\, x.\, \textbf{if } B \textbf{ then } (C;\, x) \textbf{ else skip}.$$

4.3.1 Verification of a Higher-Order Buffer

In this section, we give a last—more substantial—example of showing via a translation into the π-calculus that two implementations of a *two-place buffer* in full CIA are equivalent. Observe that the proof can be generalized to n-place buffers.

For the sake of simplicity, we assume that all buffers store integer-values. The example involves both procedures of higher order and the **await**-statement. We consider a buffer as a higher-order procedure $\mathbf{B}(x_p)$ whose argument x_p intuitively represents the clients of the buffer. This argument, $x_p(\mathbf{put}, \mathbf{get})$, is itself a higher-order procedure, taking two procedures **put** and **get** as arguments: the first one, $\mathbf{put}(x_n)$, is used by the client to store values in the buffer; the second one, $\mathbf{get}(x_r)$, is used for retrieving values from the buffer according to a FIFO strategy; x_n and x_r are identifiers of type *int* and $\mathbf{var}[int]$, respectively. Note that the **put**- and **get**-procedures are provided by the buffer, and cannot be determined by the client.

A monolithic buffer Procedure **B** below defines a *one-place buffer*; as pointed out above, x_p represents the clients, x_n denotes a value stored by a client, and x_r is a client location, to which a value retrieved from the buffer is stored. We use sugared notation for the declarations and conditionals.

$$\begin{aligned}
\mathbf{B} \stackrel{\text{def}}{=}\ & \lambda(x_p : \theta_c).\,\mathbf{new}\ [bool]\mathit{full} := \mathit{ff}, [int]\mathit{cont} := 0\ \mathbf{in} \\
& \quad (x_p\,(\lambda(x_n : int).\,\mathbf{await}\ (!\mathit{full} = \mathit{ff})\ \mathbf{then}\ (\mathit{cont} := x_n;\ \mathit{full} := \mathit{tt}))) \qquad /*\ \mathbf{put}\ */ \\
& \quad\quad (\lambda(x_r : \mathbf{var}[int]).\,\mathbf{await}\ (!\mathit{full} = \mathit{tt})\ \mathbf{then}\ (x_r := !\mathit{cont};\ \mathit{full} := \mathit{ff})). \qquad /*\ \mathbf{get}\ */
\end{aligned}$$

In our model, a one-place buffer possesses the two local variables *full* and *cont*: variable *full* indicates whether the buffer is full ($!\mathit{full} = \mathit{tt}$) and can deliver a value which is stored in *cont*, or empty ($!\mathit{full} = \mathit{ff}$) and can accept a value which it then stores in *cont*.

Analogously, one can define buffers with two, or even more, places. Buffer $\mathbf{B}_1$ below, for example, is a *two-place buffer*. It possesses the three local variables *ib*, cont_1 and cont_2. Variable *ib* tells how many values there are in the buffer: for $!ib = 0$, a value can be accepted and is stored in cont_1; for $!ib = 1$, yet another value can be accepted and is stored in cont_2, or the value in cont_1 can be delivered; for $!ib = 2$, the value in cont_1 is delivered, and that in cont_2 is shifted to cont_1.

$$\begin{aligned}
\mathbf{B}_1 \stackrel{\text{def}}{=}\ & \lambda(x_p : \theta_c).\,\mathbf{new}\ [int]ib := 0, [int]\mathit{cont}_1 := 0, [int]\mathit{cont}_2 := 0\ \mathbf{in} \\
& \quad (x_p\,(\lambda(x_n : int).\,\mathbf{await}\ (!ib \leq 1)\ \mathbf{then} \\
& \qquad\qquad ((\mathbf{if}\ (!ib = 0)\ \mathbf{then}\ \mathit{cont}_1 := x_n\ \mathbf{else}\ \mathit{cont}_2 := x_n);\ ib := !ib + 1)) \\
& \quad\quad (\lambda(x_r : \mathbf{var}[int]).\,\mathbf{await}\ (!ib \geq 1)\ \mathbf{then} \\
& \qquad\qquad (x_r := !\mathit{cont}_1;\ (\mathbf{if}\ (!ib = 2)\ \mathbf{then}\ \mathit{cont}_1 := !\mathit{cont}_2);\ ib := !ib - 1))).
\end{aligned}$$

A composite buffer Defining n-place buffers like $\mathbf{B}$ and $\mathbf{B}_1$ yields single monolithic terms. However, we can also define n-place buffers in a modular way, by connecting n one-place buffers. In this case, it is necessary to distinguish the first $n-1$ buffers from the last one, which acts as a *barrier-buffer*. For the barrier-buffer, we take the term $\mathbf{B}$ from above. To its front, we attach *head-buffers* $\mathbf{HB}(x_{pt}, x_{gt})$ obtaining the **put**- and **get**-procedures from the server-buffer. $\mathbf{HB}$ defines a **put**-procedure for the clients x_p so that they can store values into it; as the clients have to retrieve values form the server-buffer that $\mathbf{HB}$ is attached to, $\mathbf{HB}$ hands them over the **get**-procedure from the server-buffer. A head-buffer possesses the two variables full_h and cont_h, where the former indicates whether a value can be delivered ($!\mathit{full}_h = \mathit{tt}$) from cont_h or accepted ($!\mathit{full}_h = \mathit{ff}$). Whenever $\mathbf{HB}$ is full, it attempts to transmit its content to the server-buffer, using the **put**-procedure from the server. The formal definition of head-buffers $\mathbf{HB}$ in full CIA is as follows, again using sugared notation:

$$
\begin{array}{ll}
\mathbf{HB} \stackrel{\text{def}}{=} & \lambda(x_p : \theta_c).\, \lambda(x_{pt} : \mathit{int} \to \mathbf{comm}).\, \lambda(x_{gt} : \mathbf{var}[\mathit{int}] \to \mathbf{comm}). \\
& \quad \mathbf{new}\ [\mathit{bool}]\mathit{full}_h := \mathit{ff},\ [\mathit{int}]\mathit{cont}_h := 0\ \mathbf{in} \\
& \quad\quad (x_p(\lambda(x_n : \mathit{int}).\, \mathbf{await}\ (\neg!\mathit{full}_h)\ \mathbf{then}\ (\mathit{cont}_h := x_n;\ \mathit{full}_h := \mathit{tt}))) \qquad /*\ \mathbf{put}\ */ \\
& \quad\quad\quad x_{gt} \qquad /*\ \mathbf{get}\ */ \\
& \quad \|\ \mathbf{rec}\, x.\, (\mathbf{if}\ (!\mathit{full}_h)\ \mathbf{then}\ (x_{pt}(!\mathit{cont}_h);\ \mathit{full}_h := \mathit{ff};\ x)\ \mathbf{else}\ x)
\end{array}
$$

We can now define a composite two-place buffer $\mathbf{B}_2$ in terms of a head-buffer $\mathbf{HB}$ and a one-place buffer $\mathbf{B}$ by,

$$\mathbf{B}_2 \stackrel{\text{def}}{=} \lambda(x_p : \theta_c).\, \mathbf{B}\,(\mathbf{HB}\, x_p).$$

By applying conventional β-reduction, we obtain a representation of $\mathbf{B}_2$ of the form,

$$
\begin{array}{rl}
\mathbf{B}_2 = & \lambda(x_p : \theta_c).\, \mathbf{new}\ [\mathit{bool}]\ \mathit{full} := \mathit{ff},\ \mathit{full}_h := \mathit{ff},\ [\mathit{int}]\ \mathit{cont} := 0,\ \mathit{cont}_h := 0\ \mathbf{in} \\
& \quad (x_p(\lambda(x_n : \mathit{int}).\, \mathbf{await}\ (\neg!\mathit{full}_h)\ \mathbf{then}\ (\mathit{cont}_h := x_n;\ \mathit{full}_h := \mathit{tt}))) \\
& \quad\quad (\lambda(x_r : \mathbf{var}[\mathit{int}]).\, \mathbf{await}\ (!\mathit{full})\ \mathbf{then}\ (x_r :=!\mathit{cont};\ \mathit{full} := \mathit{ff})) \\
& \|\ \mathbf{rec}\, x.\, (\mathbf{if}\ (!\mathit{full}_h)\ \mathbf{then}\ ((\mathbf{await}\ (\neg!\mathit{full})\ \mathbf{then}\ (\mathit{cont} :=!\mathit{cont}_h;\ \mathit{full} := \mathit{tt}));\ \mathit{full}_h := \mathit{ff});\ x).
\end{array}
$$

Proving equality With $\mathbf{B}_1$ and $\mathbf{B}_2$ both being higher-order phrases, we do not have a means to show directly that they are observationally equivalent, let alone observationally congruent. Therefore, in order to prove $\mathbf{B}_1 \approx_{oc} \mathbf{B}_2$, we translate the phrases into the π-calculus and establish $(\nu\,\ell)(\ell.0 \mid [\![\mathbf{B}_1]\!]_p^\ell) \approx_{oc} (\nu\,\ell)(\ell.0 \mid [\![\mathbf{B}_2]\!]_p^\ell)$. Note that due to the presence of **await**, we have to use locks; hence the encoding $[\![.]\!]^\ell$.

Let $\mathbf{B}_1^{\text{body}}$ and $\mathbf{B}_2^{\text{body}}$ be the bodies of the procedures $\mathbf{B}_1$ and $\mathbf{B}_2$; they are obtained by stripping off the leading λ, which is identical for $\mathbf{B}_1$ and $\mathbf{B}_2$. With the compositionality of $\approx_\pi$, it suffices to prove that the encodings of $\mathbf{B}_1^{\text{body}}$ and $\mathbf{B}_2^{\text{body}}$ are bisimilar. We use bisimulation up to expansion; see Section 2.2.2 for an introduction. Roughly, the relation $\mathcal{R}$ used for the proof is of the following form (we omit those processes resulting from calls from clients that have not been served immediately):

$$
\begin{array}{rlll}
\mathcal{R} \stackrel{\text{def}}{=} & \{((\nu\,\ell)(\ell.0 \mid [\![\mathbf{B}_1^{\text{body}}]\!]_p^\ell), & (\nu\,\ell)(\ell.0 \mid [\![\mathbf{B}_2^{\text{body}}]\!]_p^\ell)), & \text{empty buffers} \\
& ((\nu\,\ell)(\ell.0 \mid [\![\mathbf{B}_1^{\text{body}}(v)]\!]_p^\ell), & (\nu\,\ell)(\ell.0 \mid [\![\mathbf{B}_2^{\text{body}}(v)]\!]_p^\ell)), & \text{one value stored} \\
& ((\nu\,\ell)(\ell.0 \mid [\![\mathbf{B}_1^{\text{body}}(v,w)]\!]_p^\ell), & (\nu\,\ell)(\ell.0 \mid [\![\mathbf{B}_2^{\text{body}}(v,w)]\!]_p^\ell)) & \text{two values stored} \\
& \mid\ v, w : \mathit{int}\},
\end{array}
$$

where $\mathbf{B}_1^{\text{body}}(v)$ and $\mathbf{B}_2^{\text{body}}(v)$ are like $\mathbf{B}_1^{\text{body}}$ and $\mathbf{B}_2^{\text{body}}$ but with the value v stored in them; and $\mathbf{B}_1^{\text{body}}(v,w)$ and $\mathbf{B}_2^{\text{body}}(v,w)$ are like $\mathbf{B}_1^{\text{body}}$ and $\mathbf{B}_2^{\text{body}}$ with the values v and w stored in them.

A bisimulation-proof The justification that $\mathcal{R}$ is a bisimulation up to expansion consists of the usual case-study over all pairs contained in it and all possible transitions for each of the processes involved. Consider, for instance, the first pair of the relation, where the buffers are empty; that is, $!ib = 0$ in $\mathbf{B}_1^{\text{body}}$ and $!full = !full_h = \mathit{ff}$ in $\mathbf{B}_2^{\text{body}}$. In that state the values of $cont_1$, $cont_2$, $cont$, and $cont_h$ do not matter, as they cannot be read. With corresponding sequences of transitions,

$s \;=\; x_{pt}\langle r\rangle$	client calls **put**-procedure, requesting that a value be stored, r to be used later to signal termination
$(\nu\, q)\bar{x}_n\langle q\rangle$	buffer asks for a value, with x_n a previously fixed channel, and q a fresh name to communicate value
$q\langle v\rangle$	client provides value v along q
$\bar{r}$,	buffer signals termination along r,

the buffers accept a value v from their client and, after storing it, signal the termination of that activity, thus,

$$\begin{array}{lcl} (\nu\,\ell)(\ell.0 \mid [\![\mathbf{B}_1^{\text{body}}]\!]_p^{\ell}) & \stackrel{s}{\Longrightarrow} & (\nu\,\ell)(\ell.0 \mid [\![\mathbf{B}_1^{\text{body}}(v)]\!]_p^{\ell}) \\ (\nu\,\ell)(\ell.0 \mid [\![\mathbf{B}_2^{\text{body}}]\!]_p^{\ell}) & \stackrel{s}{\Longrightarrow} & (\nu\,\ell)(\ell.0 \mid [\![\mathbf{B}_2^{\text{body}}(v)]\!]_p^{\ell}). \end{array}$$

During this execution, the buffers hold the lock; it is released at the same time the client is informed about the termination. Now, $!ib = 1$ in $\mathbf{B}_1^{\text{body}}$ and $!full = tt$ in $\mathbf{B}_2^{\text{body}}$; value v is assigned to $cont_1$ and $cont$, respectively. We can assume this, despite $\mathbf{B}_2^{\text{body}}$ first storing v in $cont_h$, as

$$(\nu\,\ell)(\ell.0 \mid [\![\mathbf{B}_2^{\text{body}}(cont_h := v)]\!]_p^{\ell}) \;\;\geq\;\; (\nu\,\ell)(\ell.0 \mid [\![\mathbf{B}_2^{\text{body}}(cont := v)]\!]_p^{\ell}),$$

where $\geq$ denotes expansion . Note that this application of the "up to" techniques which were introduced in Section 2.2.2, is vital to the proof of the example (otherwise the relation would yield an extremely large number of pairs). The other proof-obligations for $\mathcal{R}$ are similar.

In general, there are pending requests to the buffers. Some just have not been served immediately, whereas others cannot be served at once, such as a read-request when the buffers are empty, or a write-request when the buffers are full. Also, clients may request several services without waiting for one to be finished. In this case, the lock guarantees that they are served one at a time, yet in arbitrary order.

To our knowledge, there exists no technique to prove this last example within a higher-order operational framework, due to the presence of local state (compare the introduction to this chapter). We do not know how to prove this or the other examples in this section directly within the operational semantics of Algol without going through a universal quantification over contexts; recall the problems with reasoning directly within the Algol semantics, discussed above.

4.4 Discussion

The approach presented in this chapter is applicable to other languages with state. We have, for instance, modelled languages in the spirit of [MT91, MT92], using a call-by-value paradigm instead of call-by-name, and extending variables to higher order, so that also references and commands are stored in the registers.

Adding parallelism During the execution of an **await**-statement, only one thread of computation is active (see Section 4.1 and [Bro96]), yielding a purely sequential behaviour. The degree of parallelism in the presence of an active (that is, currently running) **await**-statement can be increased by, for example, a simultaneous execution of phrases which do not access variables affected by the **await**-statement. In order to increase the degree of parallelism in the presence of an active (that is, currently running) **await**-statement, one may, for instance allow for a simultaneous execution of phrases which do not access variables affected by the **await**-statement. This can be modelled, in the SOS semantics, by locks carrying along information about the concerned variables; in the π-calculus semantics, multiple locks can be introduced. The necessary information on the access to variables can be gained by some simple preliminary static analysis. Of course, such an increase in parallelism changes the overall semantics; nevertheless, there are behavioural correspondences between the more sequential and the more parallel version: first, if two phrases are bisimilar in the more parallel version, then they are also bisimilar in the sequential one (cutting off branches from the transition-systems); and, second, a phrase may yield a divergent computation (transition-trace) in the sequential semantics if and only if it does so in the parallel one (transitions occurring interleaved in the parallel semantics are *causally independent*, so they can be interchanged resulting in a computation of the sequential semantics).

Related work We follow an operational approach, as promoted for IA by Abramsky and McCusker [AM96a, AM99], and Pitts [Pit96]. Also, the work of Mason and Talcott centers towards that direction [MT91, MT92]. Most semantics apply denotational models, making use of relational parametricity to express irreversibility of state-changes; see, for instance, [Rey81, OT95, Bro96].

Completeness The issue of completeness for arbitrary phrases is delicate. Consider the following example, where P is a free identifier:

$$\begin{array}{lcl} \textbf{new}\ [int]\iota := 0\ \textbf{in} & & \textbf{new}\ [int]\iota := 0\ \textbf{in} \\ \quad P(\iota); & = & \quad P(\iota); \\ \quad \textbf{if}\ (!\iota = 0)\ \textbf{then skip} & & \quad \textbf{if}\ (!\iota = 0)\ \textbf{then}\ (\textbf{if}\ (!\iota = 1)\ \textbf{then diverge else skip}) \\ \qquad\qquad \textbf{else diverge} & & \qquad\qquad \textbf{else diverge} \end{array}$$

This example hinges on the fact that a context looses access to ι after the completion of P. This is not the case for π-calculus-translation, however. Suppose the phrase has been signalled that P has terminated, and that $\iota = 0$. One would naturally conclude that both phrases should now terminate as well. Yet, with the access gained during the execution of P, a π-calculus-context can change the value of ι into 1 even after it has once been read by the conditional; in this case, the encoding of the left-hand phrase terminates, whereas that of the right-hand phrase cannot.

For validating this example, a refined notion of linearity would be necessary, extending to variables the observer has obtained from a procedure call. Type-systems of this kind have been studied for the π-calculus in [Hon96, KPT99, Kob97]. However, even these further refinements might not suffice to obtain completeness. Moreover, our experiments have led us to the conviction that I/O-types are usually sufficient for reasoning, and that further typing would just make concrete proofs too complex.

Proof-techniques The bisimulation-proofs in this chapter make extensive use of up-to techniques and contextual arguments. In the derivation of the theoretical results in Section 4.2, we apply bisimulation up to expansion (for an introduction, see Section 2.2.2); in order to prove the example in Section 4.3.1, we further use up-to context techniques [SM92, San95, San96b]. The proofs both of theoretical results and applications are tedious, and necessitate a great amount of book-keeping; a mechanization could turn out to be advantageous. A global objective therefore is the design and implementation of an integrated platform for reasoning about and within the π-calculus. Such a framework should necessarily provide a range of up-to and algebraic proof-techniques.

Chapter 5

Mechanized Validation of Infinite-State Systems

In this chapter, we apply several bisimulation proof methods in the verification of distributed systems. In contrast to model-checking techniques, we model both *system* and *specification* in terms of processes, and apply *observation equivalence*. Observation equivalence in our case corresponds to *weak early bisimilarity* as presented in Chapter 2.

We model the systems in a *concurrent normal-form*, that is, a *system* $(C_1 \parallel \ldots \parallel C_n)$ consists of a parallel composition of a number of sequential but not necessarily deterministic *components* C_i. This kind of processes usually suffices to model reactive systems, see also [GS95]. The behaviour of the components is described by rules of the form,

$$C_i \xrightarrow{\mu} C_i',$$

possibly with side-conditions. The action μ can be an *output* $\bar{\mathbf{a}}(\tilde{\mathbf{v}})$, an *input* $\mathbf{a}(\tilde{\mathbf{v}})$, or a *silent prefix* τ. The behaviour of the compound system is determined by that of its components:

- **Composition:** if $P \xrightarrow{\mu} P'$, then $P \parallel Q \xrightarrow{\mu} P' \parallel Q$;
- **Communication:** if $P \xrightarrow{\bar{\mathbf{a}}(\tilde{\mathbf{v}})} P'$ and $Q \xrightarrow{\mathbf{a}(\tilde{\mathbf{v}})} Q'$, then $P \parallel Q \xrightarrow{\tau} P' \parallel Q'$;
- and vice versa.

We use an interleaving semantics: either one of the components performs a step which is then a step of the whole system (*composition*), or two components exchange data so the whole system will indicate by a τ that something is going on internally, but no further information is passed to the observer (*communication*). Like in the previous chapters,

$$P \stackrel{\epsilon}{\Longrightarrow} P' \quad \stackrel{\text{def}}{=} \quad P(\xrightarrow{\tau})^* P'$$

denotes the reflexive-transitive closure of strong internal transitions, and

$$P \stackrel{\mu}{\Longrightarrow} P' \quad \stackrel{\text{def}}{=} \quad P \stackrel{\epsilon}{\Longrightarrow}\xrightarrow{\mu}\stackrel{\epsilon}{\Longrightarrow} P'$$

yields weak versions of transitions between one state of a component or a system, and another. The following correspondence between strong and weak transitions can be derived, see also Sections 2.1.1 and 2.3.1:

- if $P \xrightarrow{\mu} P'$, then $P \overset{\mu}{\Longrightarrow} P'$;
- if $P \overset{\tau}{\Longrightarrow} P'$, then $P \overset{\epsilon}{\Longrightarrow} P'$;
- **Composition:** if $P \overset{\mu}{\Longrightarrow} P'$, then $P \parallel Q \overset{\mu}{\Longrightarrow} P' \parallel Q$;
- **Communication:** if $P \overset{\bar{\mathbf{a}}(\tilde{\mathbf{v}})}{\Longrightarrow} P'$ and $Q \overset{\mathbf{a}(\tilde{\mathbf{v}})}{\Longrightarrow} Q'$, then $P \parallel Q \overset{\tau}{\Longrightarrow} P' \parallel Q'$;
- and vice versa.

The derivation is by a standard induction on the length of the transitions, and can easily be mechanized in a theorem-prover, see Section 2.3.1. In contrast to similar derivations for the π-calculus in Chapter 3, no β-abstractions in the sense of Theorem 3.14 is necessary, because syntax and semantics are essentially first-order. Nevertheless, no notion of substitution has to be defined, because the syntax refers to *concrete* systems in contrast to the *abstract* description of the π-calculus before.

The chapter is divided into three sections: in Section 5.1, we introduce a generalization of observation equivalence which allows us to omit restrictions, so we can circumvent the question of binders. It is based on providing a set of admitted names L, hence we call it *observation equivalence wrt. L*. We apply it in two case studies from the field of communication protocols, one dealing with faulty channels (see Section 5.1.1) and the other validating the Alternating Bit Protocol (ABP, see Section 5.1.2). We pursue the semantic approach in that we exhibit concrete bisimulation-relations. The two examples demonstrate the support general-purpose therom-provers can offer in the verification of infinite-state systems using bisimulations; in particular, the whole book-keeping is dealt with by Isabelle. Further, in some cases the automatic tactics succeed quite well by themselves when trying to find a matching weak derivative in the single proof-obligations. Then we discuss proof techniques for observation equivalence (wrt. L). In Section 5.2, we apply compositionality, so the verification of a compound system can be tackled in a divide-and-conquer manner: first validate the components, and then use their specifications in the validation of the entire system. We do this an a proof about the specification of a Sliding-Window Protocol (SWP, see Section 5.2.1). In Section 5.3, we apply an *up-to* proof-technique using *bisimulations up to expansion* in order to reduce the size of the relation. This is achieved by detecting classes of states within every system that are observation-equivalent. Out of these, we select the 'smallest' process P, in the sense that it is the most efficient one. For each class, we consider only such a P in the bisimulation-relation, omitting all the others. The equivalence classes plus their representatives are derived in separate proofs. Like this, we still consider the entire state-spaces of system and specification, yet in a more modular way which results in separate tractable proofs instead of one intractable (because of being too large) argument. We apply both a standard and an *up-to* bisimulation in the validation of a simple cache-protocol, so we can compare the two approaches (see Section 5.3.1).

Remark: All material presented in this chapter has been formalized in Isabelle/HOL; the proof-scripts are available at `http://www7.in.tum.de/~roeckl/thesis/protocols/` (communication-protocols) and `http://www7.in.tum.de/~roeckl/Cache/` (memory-cache-coherence). The proofs are also presented in [RE99, Röc00, RE00].

5.1 Observation-Equivalence wrt. L

We use a variation of observation-equivalence as introduced by Milner and Park [Mil89, Par80], which is parameterized over a set of names telling those names that can be visible to an observer from those that can only be used in internal communications. The standard notion of observation equivalence can then be obtained by assuming the associated set of names to be $\mathcal{N}$. Using a set L of *admissible* visible names, we do not have to model restrictions, because our systems are given in a concurrent normal-form (see introduction to this chapter), and an outer restriction $P\backslash A$ can be captured by the set of admissible names $\mathcal{N} - A$. This avoidance of restrictions is particularly convenient in a mechanization of bisimulation proofs, because it makes the decision superfluous whether to formalize a restriction operator in first-order or higher-order syntax. As pointed out in Chapters 2 and 3 of this thesis, either choice would bear hard-to-deal-with consequences, in the form of either substitutions or a measure to rule out extic terms, for instance, well-formedness predicates.

DEFINITION 5.1 (OBSERVATION EQUIVALENCE WRT. L) A relation $\mathcal{R} \subseteq \mathcal{P} \times \mathcal{P}$ is a *weak bisimulation wrt.* $L \subseteq \mathcal{N}$, if for all $P\mathcal{R}Q$, all $\alpha \in \mathcal{N} \cup \overline{\mathcal{N}} \cup \{\tau\}$, and all $P', Q' \in \mathcal{P}$,

(i) if $P \stackrel{\alpha}{\longrightarrow} P'$, there exists a Q' such that $Q \stackrel{\hat{\alpha}}{\Longrightarrow} Q'$ and $P'\mathcal{R}Q'$.

(ii) if $Q \stackrel{\alpha}{\longrightarrow} Q'$, there exists a P' such that $P \stackrel{\hat{\alpha}}{\Longrightarrow} P'$ and $P'\mathcal{R}Q'$.

Observation equivalence wrt. L is the union $\approx_L \stackrel{\text{def}}{=} \bigcup\{\mathcal{R} \mid \mathcal{R} \text{ is a weak bisimulation wrt. } L\}$.

In this definition, the set of names $\mathcal{N}$ is used in a slightly different manner from the π-calculus. We consider every input $\mathbf{a}(\tilde{x})$, for a sequence of values $\tilde{x}$, as a name in $\mathcal{N}$; the set of co-names $\overline{\mathcal{N}}$ is given by the corresponding outputs $\bar{\mathbf{a}}(\tilde{x})$. This implies that the set of visible labels $\mathcal{L}$ is obtained from a unification of names and co-names, $\mathcal{N} \cup \overline{\mathcal{N}}$, like in pure CCS [Mil89].

Observation equivalence wrt. L corresponds naturally with standard observation equivalence: whenever two processes are equivalent wrt. some set L, their restrictions with the complement of L are observation equivalent; further, standard observation equivalence implies observation equivalence wrt. arbitrary L, because of well-known congruence results.

PROPOSITION 5.2 *For all $P, Q \in \mathcal{P}$ and all $L \subseteq \mathcal{N}$, the following holds:*

1. *$P \approx_L Q$ implies $P\backslash(\mathcal{N} - L) \approx Q\backslash(\mathcal{N} - L)$;*

2. *$P \approx Q$ implies $P \approx_L Q$.*

Proof: 1. Let $\mathcal{R}$ be a weak bisimulation wrt. L. Then the relation

$$\mathcal{R}' \stackrel{\text{def}}{=} \{ (P\backslash(\mathcal{N} - L),\, Q\backslash(\mathcal{N} - L)) \mid (P, Q) \in \mathcal{R} \}$$

is a weak bisimulation.

2. Owing to well-known congruence results, $P \approx Q$ implies $P\backslash(\mathcal{N} - L) \approx Q\backslash(\mathcal{N} - L)$. Then, from the weak bisimulation $\mathcal{R}$ proving $P \approx Q$, one can construct a weak bisimulation

$$\mathcal{R}' \stackrel{\text{def}}{=} \{ (P\backslash(\mathcal{N} - L),\, Q\backslash(\mathcal{N} - L)) \mid (P, Q) \in \mathcal{R} \}$$

K1	$K(s) \xrightarrow{\text{acc}(x)} K(x_c s)$	**L1**	$L(s) \xrightarrow{\text{acc}(x)} L(xs)$	(accept)
K2	$K(sx_b t) \xrightarrow{\tau} K(st)$	**L2'**	$L(sxt) \xrightarrow{\tau} L(st)$	(lose)
K3	$K(sx_b t) \xrightarrow{\tau} K(sx_b x_b t)$	**L3**	$L(sxt) \xrightarrow{\tau} L(sxxt)$	(duplicate)
K4	$K(sx_b t) \xrightarrow{\tau} K(sx_g t)$			(garble)
K5	$K(sx_b) \xrightarrow{\overline{\text{d}}(x_b)} K(s)$	**L5**	$L(sx) \xrightarrow{\overline{\text{del}}(x)} L(s)$	(deliver)
F1	$F \xrightarrow{\text{d}(x_c)} F(x)$	**F1'**	$F \xrightarrow{\text{d}(x_g)} F$	(accept/discard)
F2	$F(x) \xrightarrow{\tau} F$	**F2'**	$F(x^{n+1}) \xrightarrow{\tau} F(x^n)$	(lose, $n > 0$)
F3	$F(x^n) \xrightarrow{\tau} F(x^{n+1})$			(duplicate)
F4	$F(x) \xrightarrow{\overline{\text{del}}(x)} F$	**F4'**	$F(x^{n+1}) \xrightarrow{\overline{\text{del}}(x^n)} F$	(deliver, $n > 0$)

Table 5.1: Faulty Communication Channels. An implementation K can lose, duplicate, and garble accepted messages (**K1**–**K5**). We assume garbling of messages to be detectable. A bit b attached to each message tells whether the message is still correct ($b = c$) or has been garbled ($b = g$). In combination with a filter F (**F6**–**F9**, **F6'**–**F9'**), it behaves like a specification L (**L1**–**L5**).

for which (by simply rewriting its definition)

$$\mathcal{R}'' \stackrel{\text{def}}{=} \{ (P, Q) \mid (P \backslash (\mathcal{N} - L),\, Q \backslash (\mathcal{N} - L)) \in \mathcal{R}' \}$$

is a weak bisimulation wrt. L. □

In the following two sections, we use observation equivalence wrt. a set of names for a mechanical validation of two benchmark examples concerning communication protocols. We follow the *semantic* approach, exhibiting a relation and proving that it is a bisimulation. This contrasts the *syntactic* approach, that has been used more widely (see, for instance, [GS95, BG88, PS91]), in which an axiomatization of the applied equivalence is given and used to transform the system into its specification. While the semantic approach requires a good intuition of the behaviour of the systems, the syntactic approach necessitates a profound understanding of the proof-system. A possible advantage of the semantic approach lies in the detection of errors in the description of the system. Whereas a transformation simply does not yield the specification, an analysis of the behaviour in a bisimulation proof exhibits exactly those states with unintended behaviour. This allows one to modify the behaviour of exactly those states successively, and finally obtain a correct system.

5.1.1 Faulty Channels

Our first example is taken from [Sne95]. It is of interest to us for the following reasons: (1) it compares two nondeterministic infinite-state systems operating on similar data structures, and (2) for most of the resulting proof obligations it suffices to find matching strong transitions, that is, in most cases an action by one of the processes is matched by one single action of the other processes. Reason (1) emphasizes that the bisimulation proof method is universally applicable, and does not require, for instance, the specification to be deterministic. Reason (2) allows Isabelle/HOL to conduct the proof almost automatically.

Consider two channels, K and L, of unbounded capacity; their contents can be modelled by finite lists of arbitrary length, and so we denote a state of channel K by $K(x_1 \dots x_n)$, where $x_1 \dots x_n$ is a list of messages, each of them taken from an arbitrary domain M of possible messages. This can be modelled by using a type variable, because Isabelle supports polymorphism. K and L are simple components, that is, processes without parallel composition; Table 5.1 contains the rules formally describing their behaviour. Both K and L may lose (**K2**, **L2**) or duplicate (**K3**, **L3**) messages, but K is further able to garble data (**K4**). We suppose that it is possible to detect whether a message is correct or garbled; therefore, all messages in K are marked with an index c if they are correct, and g if they are garbled; b stands for either c or g. In order to sort out garbled messages at the end of K, we use a filter F which, when attached to a channel, delivers correctly transmitted messages (**F1**, **F4**, **F4'**) and discards garbled ones (**F1'**). Like the channels, F is itself faulty: it can lose (**F2**, **F2'**) or duplicate (**F3**) messages. We prove that the parallel composition of K and F—that is, $K \parallel F$—is observationally equivalent to L with respect to the set of names $L_1 = \{\, \mathrm{acc}(x), \mathrm{del}(x) \mid x \in M \,\}$. (This is not so trivial as it may seem; for instance, it does not hold if F cannot lose messages, see the remark below.) For this, we show that the relation

$$\begin{array}{rlll} \mathcal{F} & \stackrel{\text{def}}{=} & \{\, (K(s) \parallel F, \quad L(\hat{s})) & \mid \ s \text{ an indexed sequence of messages,} \\ & & & \quad \hat{s} = s \text{ without indices and garbage} \,\} \\ & \cup & \{\, (K(s) \parallel F(x^n),\ L(\hat{s}x^n)) & \mid \ s \text{ an indexed sequence of messages,} \\ & & & \quad \hat{s} = s \text{ without indices and garbage,} \\ & & & \quad x \text{ a message} \,\} \end{array}$$

is a bisimulation wrt. L_1, where $\hat{s}$ in L are obtained from lists s in K by first eliminating all garbled messages and then clearing all remaining messages of their c tags. For a list $s = a_c a_g a_c b_c b_c c_g a_c$, for example, $\hat{s} = aabba$.

Remark: We have considered a faulty filter F which loses and duplicates messages itself. For a correct filter F', there is no result $K \parallel F' \approx_{L_1} L$. Intuitively, if the filter is correct then $K \parallel F'$ is more reliable than L, while observation equivalence requires the two systems to have the same degree of (un)reliability. In this case, one can use *observation preorder*, in which L has to simulate the behaviour of $K \parallel F'$ but not vice versa.

Evaluation Proving that $\mathcal{F}$ is a bisimulation is not difficult, and most of the involvement of the user goes into theorems describing, for instance, the relation between $\hat{s}$ and $\hat{s}'$ if s' is obtained from s by losing one message. Provided with these theorems, Isabelle proves by one single application of `auto_tac` that $\mathcal{F}$ is a bisimulation, automatically guessing, for instance, the matching weak transitions. The proof script contains less than 300 lines, and has been set up within a few hours only.

5.1.2 The Alternating Bit Protocol

The Alternating Bit Protocol (ABP), proposed in [Lyn68, BSW69], is a well-established benchmark for proof methodologies implemented in theorem provers (see, for instance, [PS91, BG88, NS94, Gim96]). It turns unreliable channels into reliable communication lines. We consider an infinite-state variant in which the channels can hold arbitrarily many messages, like in the

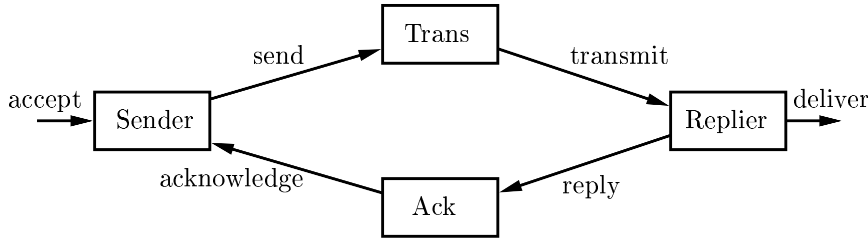

Figure 5.1: **Components of the ABP.** The protocol is designed to turn the unreliable channels `Trans` and `Ack` into a reliable communication line.

previous example. We assume that messages can be lost and duplicated, but not garbled. In order to construct a protocol also dealing with (detectable) garbling we can use the result of the previous subsection, replacing a channel by the parallel composition of a channel and a filter. Compositionality of observation equivalence automatically guarantees the correctness of this new protocol.

The model as well as the outline of the proof follow [Mil89]. The protocol is modelled as the parallel composition of four processes; it is schematically depicted in Figure 5.1. Messages and acknowledgements are transmitted over two unreliable channels `Trans`, or T for short, and `Ack`, or A for short, by a *sender* S, and a *replier* module R. The intended behaviour of the protocol can be specified by a one-place buffer B. As pointed out by Milner [Mil89], this allows to abstract from the data to be transmitted. We follow this argument, omitting concrete messages in our formalization. Note, however, that including them would not further complicate the proof, especially not in a theorem-prover.

The channels T and A exhibit exactly the same (faulty) behaviour as channel L in Section 5.1.1: they have unbounded capacity, but can lose or duplicate messages at any time. Initially, we assume both channels to be empty. The *sender* module continuously accepts messages from the environment (**S1**), transmits them repeatedly over channel T (**S2**, **S3**) and waits for the current acknowledgement along A (**S4**, **S5**), before accepting a new message. After having delivered a message to the environment (**R1**), the replier R repeatedly transmits tagged acknowledgements to the sender (**R2**, **R3**) until a new message arrives (**R4**, **R5**). The initial state of the protocol is the process,

$$ABP = S_a(0) \parallel T(\epsilon) \parallel A(\epsilon) \parallel R_s(1) \, .$$

Table 5.2 gives a formal description of the behaviour of the components.

As indicated above, we intend the system to behave like a buffer of capacity one. This specification is also formalized in Table 5.2, the initial state being B_a. We show that $ABP \approx_{L_1} B_a$. Recall from the previous section that $L_1 = \{\text{acc}, \text{del}\}$, only that we now abstract from the data because there is never more than one current message in the system.

The states of the protocol can be divided into two classes: a message can either be accepted or delivered, possibly after a finite number of silent transitions. The bisimulation relation $\mathcal{B}$

System: transmission channels

T1	$T(s) \xrightarrow{\mathrm{cs}(x)} T(xs)$	**A1**	$A(s) \xrightarrow{\mathrm{cr}(x)} A(xs)$	(accept)
T2	$T(sxt) \xrightarrow{\tau} T(st)$	**A2**	$A(sxt) \xrightarrow{\tau} A(st)$	(lose)
T3	$T(sxt) \xrightarrow{\tau} T(sxxt)$	**A3**	$A(sxt) \xrightarrow{\tau} A(sxxt)$	(duplicate)
T4	$T(sx) \xrightarrow{\overline{\mathrm{ct}}(x)} T(s)$	**A4**	$A(sx) \xrightarrow{\overline{\mathrm{ca}}(x)} A(s)$	(transmit)

System: sender and receiver

S1	$S_a(b) \xrightarrow{\mathrm{acc}} S_s(b)$	**R1**	$R_a(b) \xrightarrow{\overline{\mathrm{del}}} R_s(b)$	(accept/deliver)
S2	$S_s(b) \xrightarrow{\overline{\mathrm{cs}}(b)} S_w(b)$	**R2**	$R_s(b) \xrightarrow{\overline{\mathrm{cr}}(b)} R_w$	(transmit)
S3	$S_w(b) \xrightarrow{\tau} S_s(b)$	**R3**	$R_w(b) \xrightarrow{\tau} R_s(b)$	(timeout)
S4	$S_w(b) \xrightarrow{\mathrm{ca}(b)} S_a(\neg b)$	**R4**	$R_w(b) \xrightarrow{\mathrm{ct}(\neg b)} R_a(\neg b)$	(receive new copy)
S5	$S_w(b) \xrightarrow{\mathrm{ca}(\neg b)} S_w(b)$	**R5**	$R_w(b) \xrightarrow{\mathrm{ct}(b)} R_w(b)$	(receive old copy)

Specification: one-place buffer

B1	$B_a \xrightarrow{\mathrm{acc}} B_d$	**B2**	$B_d \xrightarrow{\overline{\mathrm{del}}} B_a$	(accept/deliver)

Table 5.2: **Implementation and Specification of the ABP.** Sender S (**S1**–**S5**) and Replier R (**R1**–**R5**) repeatedly transmit copies of the current message (acknowledgement) until they receive an acknowledgement (a new message). Old copies are distinguished from new ones with the help of an alternating bit b. The ABP is supposed to turn faulty channels of arbitrary size into reliable one-place buffers B (**B1**–**B2**). The indices refer to the states S, R, and B can assume.

thus falls into the two corresponding sub-relations

$$
\begin{array}{lcll}
\mathcal{B}_a & \stackrel{\mathrm{def}}{=} & \{ & (S_a(\neg b) \parallel T(b^n) \parallel A(b^p) \parallel R_s(b), \quad B_a) \\
& & & (S_a(\neg b) \parallel T(b^n) \parallel A(b^p) \parallel R_w(b), \quad B_a) \\
& & & (S_s(\neg b) \parallel T((\neg b)^n) \parallel A(b^p(\neg b)^q) \parallel R_s(\neg b), \quad B_a) \\
& & & (S_s(\neg b) \parallel T((\neg b)^n) \parallel A(b^p(\neg b)^q) \parallel R_w(\neg b), \quad B_a) \\
& & & (S_w(\neg b) \parallel T((\neg b)^n) \parallel A(b^p(\neg b)^q) \parallel R_s(\neg b), \quad B_a) \\
& & & (S_w(\neg b) \parallel T((\neg b)^n) \parallel A(b^p(\neg b)^q) \parallel R_w(\neg b), \quad B_a) \mid b \in \{0,1\} \},
\end{array}
$$

capturing those states eventually accepting a new message, and

$$
\begin{array}{lcll}
\mathcal{B}_d & \stackrel{\mathrm{def}}{=} & \{ & (S_s(\neg b) \parallel T((\neg b)^m) \parallel A(b^p) \parallel R_a(\neg b), \quad B_d) \\
& & & (S_w(\neg b) \parallel T((\neg b)^m) \parallel A(b^p) \parallel R_a(\neg b), \quad B_d) \\
& & & (S_s(\neg b) \parallel T((\neg b)^m b^n) \parallel A(b^p) \parallel R_s(\neg b), \quad B_d) \\
& & & (S_s(\neg b) \parallel T((\neg b)^m b^n) \parallel A(b^p) \parallel R_w(\neg b), \quad B_d) \\
& & & (S_w(\neg b) \parallel T((\neg b)^m b^n) \parallel A(b^p) \parallel R_s(\neg b), \quad B_d) \\
& & & (S_w(\neg b) \parallel T((\neg b)^m b^n) \parallel A(b^p) \parallel R_w(\neg b), \quad B_d) \mid b \in \{0,1\} \},
\end{array}
$$

where eventually a message will be delivered. In every channel, there are at most two types of messages or acknowledgements, each type coming in a consecutive block. The finite lists in T and A are thus either of the form x^n, or $x^n y^m$; in Isabelle this can be expressed in terms of the `replicate`-operator from the theory for finite lists delivered with Isabelle/HOL.

To show that $\mathcal{B} \stackrel{\mathrm{def}}{=} \mathcal{B}_a \cup \mathcal{B}_d$ is indeed a bisimulation wrt. (an abstraction of) $L_1 = \{\mathrm{acc}, \mathrm{del}\}$, we follow our usual scheme. As a typical example, consider the case where the ABP performs

a strong acc-transition. We have to prove the obligation, if $(P, Q) \in \mathcal{B}$ and $P \xrightarrow{acc} P'$, then there exists a state Q' such that $Q \stackrel{acc}{\Longrightarrow} Q'$, and $(P', Q') \in \mathcal{B}$. Out of the six subrelations in $\mathcal{B}_a$ differing in the shape of P, Isabelle automatically extracts the first two as those in which P can actually do an acc. It remains to show that in both cases the resulting process P' fits the shape of the left process in the third and fourth subrelations of $\mathcal{B}_d$. The difficulty of this proof-step results from the lists in T and A looking differently in $\mathcal{B}_a$ and $\mathcal{B}_d$. Once provided with the necessary theorems about finite lists, however, Isabelle completes the argument fully automatically.

Another interesting example is the reverse case, in which $Q \xrightarrow{acc} Q'$ and $P \stackrel{acc}{\Longrightarrow} P'$. For the third through sixth case of $\mathcal{B}_a$, the user has to provide suitable sequences of weak transitions leading to the acceptance of a new message. In all of the cases, we can apply the following scheme: remove all messages and acknowledgements from T and A (that this is possible can be shown once in a separate proof, by an induction on the length of the lists stored in the channels), then have R transmit an acknowledgement to S, and finally execute the acc-transition.

For the invisible transitions of the ABP, we essentially have to show that their derivatives still lie within $\mathcal{B}_a$ or $\mathcal{B}_d$, respectively. As for each of the processes there are several possibilities, we examine each of the twelve cases in separate proofs. A simultaneous treatment of all the cases may exceed the capacity of Isabelle's automatic tactics, as also the hypothetical cases like "component A communicates with component T" have to be considered, resulting in an exponential blow-up of cases. Again, provided with the necessary theorems about lists, Isabelle proves the cases fully automatically.

Evaluation The proof-script contains about 800 lines. As nearly half of it consists of theorems about the finite lists used in the channels, some experience with theorem-provers is necessary to set up the proofs. The bisimulation part itself can be set up within a few days by a user experienced both in the bisimulation proof-method and theorem-proving. In particular, only a few proof procedures strongly based on Isabelle's automatic tactics are necessary to capture all of the almost 100 proof-obligations. Note that, as pointed out by Milner [Mil89], this example is clearly on the edge of what can be proved without machine-assistance, if not beyond.

5.2 Compositionality

A major characteristic of observation equivalence is that it is *compositional*, yielding a divide-and-conquer verification strategy: In order to verify a compound system $Sy_1 \parallel \ldots \parallel Sy_n$ with respect to a specification Sp, one verifies the single components Sy_i with respect to specifications Sp_i and then compares $Sp_1 \parallel \ldots \parallel Sp_n$ with Sp. As a consequence, the sizes of the single bisimulation relations are drastically reduced. The theoretical result underlying this proof strategy is that for all $P, Q, R \in \mathcal{P}$, an equivalence $P \approx Q$ implies $P \parallel R \approx Q \parallel R$, and, similarly, $P \approx_L Q$ implies $P \parallel R \approx_L Q \parallel R$.

We have already applied compositionality in the previous section, arguing that instead of channels that lose, duplicate, and garble messages, we can use channels that only lose or duplicate messages, in our model of the ABP. As a result, we have obtained a smaller description which is easier to verify. In the following section, we are going to apply compositionality systematically in several steps of a bisimulation proof showing the correctness of a specification of the Sliding-Window Protocol (SWP).

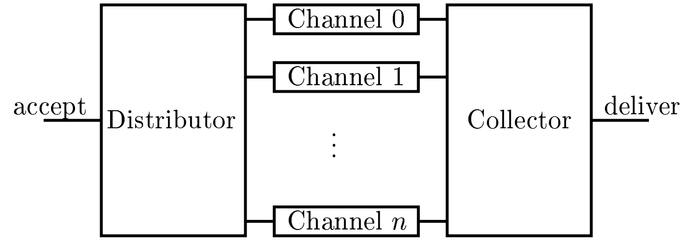

Figure 5.2: **A Specification of the SWP.** The parameterized protocol consists of a parallel composition of n (pairs of) communication lines implementing the ABP; the ABP is used to guarantee that a message handed over to one of the communication lines is transmitted reliably.

System: distributor and collector

D1	$D_n^h \xrightarrow{\mathrm{acc}(x)} D_n^h(x)$	**C1**	$C_n^l \xrightarrow{\mathrm{co}(x)} C_n^{l+_n 1}(x)$	(accept/collect)
D2	$D_n^h(x) \xrightarrow{\overline{\mathrm{ci}}_h(x)} D_n^{h+_n 1}$	**C2**	$C_n^l(x) \xrightarrow{\overline{\mathrm{del}}(x)} C_n^l$	(distribute/deliver)

Specification: buffer of capacity $n+2$

B3	$B^{n+2}(s) \xrightarrow{\mathrm{ci}(x)} B^{n+2}(sx)$	if $\lvert s \rvert < n+2$	(accept)
B4	$B^{n+2}(xs) \xrightarrow{\mathrm{co}(x)} B^{n+2}(s)$		(deliver)

Table 5.3: **Components of the SWP.** Rules **D1** and **D2** describe the behaviour of the distributor, **C1** and **C2** that of the collector components of the SWP. The distributor D_n continuously accepts values from the environment and transmits them to consecutive communication lines; the collector C_n recollects the values from the communication lines and delivers them to the recipient. Owing to the cyclic behaviour of both D_n and C_n, values are always transmitted in the right order. An SWP with n channels can be specified by an $n+2$-place buffer, as described by rules **B3** and **B4**.

Another property that can be important in equivalence proofs, especially when using induction, is associativity of parallel composition; that is, for all $P, Q, R \in \mathcal{P}$, it holds that $(P \parallel Q) \parallel R \approx P \parallel (Q \parallel R)$, respectively, $(P \parallel Q) \parallel R \approx_L P \parallel (Q \parallel R)$. Induction is suitable in order to show that a single component P_s models the parallel composition of n processes of equal structure, $P \parallel \ldots \parallel P$. In order to be free whether to attach such components to the front or to the back—or even both—associativity of parallel composition can be exploited.

5.2.1 A Sliding-Window Protocol

In [PS91], a simple parameterized Sliding Window Protocol (SWP) with input and output windows of equal size is presented. The system consists of n communication lines each of which uses the ABP on faulty channels. Figure 5.2 shows a schematical view. Incoming messages are cyclically distributed to the communication lines by a *distributor* D_n, and are recollected and delivered by a *collector* C_n; their behaviour is formally described in Table 5.3 (**D1**, **D2**, and **C1**, **C2**). The initial state of the SWP is the process

$$SWP \quad = \quad D_n^0 \parallel ABP_1 \parallel \ldots \parallel ABP_n \parallel C_n^0$$

where ABP_i is a copy of the process of the previous section with the acc- and del-actions

B5	$B^A(s) \xrightarrow{\mathrm{ci}_h(x)} B^A(s[h := x])$	if $s!h = \Box$	(accept)
B6	$B^A(s) \xrightarrow{\overline{\mathrm{co}}_h(x)} B^A(s[h := \Box])$	if $s!h = x \neq \Box$	(deliver)

Table 5.4: **An array-buffer.** The buffer B^A consists of an array of length n, the cells of which either contain some value x, or an auxiliary symbol $\Box$ to denote emptiness. It models the parallel composition of n one-place buffers.

suitably renamed. The system should behave like a buffer B^{n+2} of capacity $n+2$, with n being the number parallel channels in the system (the capacity is $n+2$ and not n because the distributor and the collector contribute with one place each), see also Table 5.3 (**B1**, **B2**). This time we cannot abstract from data, as the system does not necessarily need to deliver a message before accepting a new one: we have to guarantee that messages not be swapped. So we prove $SWP \approx_{L_1} B^{n+2}$ where $L_2 = \{\, \mathrm{acc}(x), \overline{\mathrm{del}}(x) \mid x \in M \,\}$. The proof falls into several parts, and makes extensive use of compositionality.

Replace the ABPs In a first step, we can apply compositionality to replace the ABP-components with one-place buffers, due to the equivalence result of the previous section:

$$(D_n^0 \parallel ABP_1 \parallel \ldots \parallel ABP_n \parallel C_n^0) \quad \approx_{L_1} \quad (D_n^0 \parallel B_{a1} \parallel \ldots \parallel B_{an} \parallel C_n^0).$$

As a result, we can observe that we have got rid of the infinite-state ABP-processes using two-state components instead. Yet, we still have to carry out a parametric proof, valid for all n. In the sequel, we further exploit extensionality in order to finally replace $B_{a1} \parallel \ldots \parallel B_{an}$ by an n-place buffer into which we then integrate D_n and C_n.

An array-buffer We need a finite representation of the n one-place buffers $B_{a1} \parallel \ldots \parallel B_{an}$ put in parallel. As a matter of design, we describe them in terms of a single component B^A, which contains an array of n cells that are initially empty; we refer to B^A as an *array-buffer*. We use an auxiliary element $\Box$ to denote that a cell in an array-buffer is empty. Table 5.4 gives a formal description of the behaviour of B^A. We show that $B^A(\Box^n) \approx_{L_2} (B_{a1} \parallel \ldots \parallel B_{an})$, with $L_2 = \{\, \mathrm{ci}_h(x), \mathrm{co}_h(x) \mid h \in I\!N,\ x \in M \,\}$. The proof combines induction with the bisimulation proof method described above. We show that for each n, the relation

$$\begin{aligned} \mathcal{B}_{ab}(n) \;\stackrel{\mathrm{def}}{=}\; & \{\, (B_a \parallel B^A(s),\ B^A(s\Box)) \;\mid\; s \in M^n \,\} \\ \cup\; & \{\, (B_d(x) \parallel B^A(s),\ B^A(sx)) \;\mid\; x \in M,\ s \in M^n \,\} \end{aligned}$$

is a bisimulation wrt. L_2. Then, by compositionality, we obtain,

$$(D_n^0 \parallel B_{a1} \parallel \ldots \parallel B_{an} \parallel C_n^0) \quad \approx_{L_1} \quad (D_n^0 \parallel B^A(\Box^n) \parallel C_n^0).$$

We can observe that values are stored in and retrieved form B^A in a cyclic order, because D_n and C_n obey a cyclic behaviour: when the distributor in state $D_n^h(x)$ encounters that the current cell numbered h is free to store value x, it does so and proceeds in state $D_n^{h+_n 1}$; similarly, when C_n^l detects a value in cell l, it reads the value to proceed in state $C_n^{l+_n 1}(x)$; see also Table 5.3. Note that by obeying a synchronous communication paradigm, it is guaranteed that values are only transmitted to empty communication lines, and are only read from full ones.

B7	$B^P(h,l,s) \xrightarrow{\mathrm{ci}(x)} B^P(h +_{\lvert s\rvert} 1, l, s[h := x])$	if $s!h = \square$	(accept)
B8	$B^P(h,l,s) \xrightarrow{\overline{\mathrm{co}}(x)} B^P(h, l +_{\lvert s\rvert} 1, s[l := \square])$	if $s!l = x \neq \square$	(deliver)

Table 5.5: **A cyclic buffer.** The buffer B^P consists of an array of length n, like the array-buffer from Table 5.4: each cell either contains an element x, or $\square$ to denote that it is empty. The cells are not independent, however, values are rather stored and retrieved in cyclic order. We use $+_m$ for addition modulo m.

A cyclic n-place buffer We emphasize the cyclic order in which values are transmitted by modelling a *cyclic* n-place buffer B^P storing values in and retrieving them from consecutive cells; its behaviour is formally described in Table 5.5. As now cyclicity is guaranteed by the buffer itself, it suffices to attach a barrier one-place buffers B_a to its front and one to its back. We can show that $B_a \parallel B^P(\square^n) \parallel B_a$ models $D_n^0 \parallel B^A(\square^n) \parallel C_n^0$, that is,

$$(D_n^0 \parallel B^A(\square^n) \parallel C_n^0) \quad \approx_{L_1} \quad (B_a \parallel B^P(0,0,\square^n) \parallel B_a),$$

and hence the SWP, again by exhibiting a suitable bisimulation wrt. L_1,

$$\begin{array}{rcl}
\mathcal{B}_{cb} & \stackrel{\mathrm{def}}{=} & \{\,(D_n^h \parallel B^A(s) \parallel C_n^l,\ B_a \parallel B^P(h,l,s) \parallel B_a) \mid s \in M^n\,\} \\
& \cup & \{\,(D_n^h(x) \parallel B^A(s) \parallel C_n^l,\ B_d(x) \parallel B^P(h,l,s) \parallel B_a) \mid x \in M,\ s \in M^n\,\} \\
& \cup & \{\,(D_n^h \parallel B^A(s) \parallel C_n^l(y),\ B_a \parallel B^P(h,l,s) \parallel B_d(y)) \mid y \in M,\ s \in M^n\,\} \\
& \cup & \{\,(D_n^h(x) \parallel B^A(s) \parallel C_n^l(y),\ B_d(x) \parallel B^P(h,l,s) \parallel B_d(y)) \mid x, y \in M,\ s \in M^n\,\}.
\end{array}$$

The relation $\mathcal{B}_{cb}$ falls into four sub-relations: in the first one, distributor and collector as well as the corresponding barrier buffers are empty; in the second one, distributor and the corresponding buffer have accepted a message x; in the third one, collector and the corresponding buffer can deliver a message y; and in the fourth one, there are messages in all barrier components. The lists s in the B^A and B^P components always correspond. Note further that the positions of h and l always correspond, too.

An $n+2$-place buffer To complete the proof, we still have to show by exhibiting suitable bisimulations that B^P with its two one-place buffers behaves like the $n+2$-buffer from the specification from Table 5.3, that is,

$$(B_a \parallel B^P(0,0,\square^n) \parallel B_a) \quad \approx_{L_1} \quad B^{n+2}.$$

In a first proof, we show that B^P behaves like B^n, and in a second proof, we justify that for every n, the buffer B^n behaves like a parallel composition of n one-place buffers B_a. The bisimulation wrt. L_2 for the first proof is,

$$\begin{array}{rcl}
\mathcal{B}_{nb}(n) & \stackrel{\mathrm{def}}{=} & \{\,(B^P(h,l,\square^l s \square^{n-h}),\ B^n(s)) \mid s \in M^{h-l}\,\} \\
& \cup & \{\,(B^P(h,l,s\square^{l-h}t),\ B^n(s)) \mid s \in M^h, t \in M^{n-l}\,\}.
\end{array}$$

That for the second proof is,

$$\begin{array}{rcl}
\mathcal{B}_b(n) & \stackrel{\mathrm{def}}{=} & \{\,(B_a \parallel B^n(s),\ B^{n+1}(s)) \ \mid\ s \in M^k,\ 0 \leq k < n\,\} \\
& \cup & \{\,(B_d(x) \parallel B^n(s),\ B^{n+1}(sx)) \ \mid\ x \in M,\ s \in M^k,\ 0 \leq k < n\,\},
\end{array}$$

applying an inductive argument similar to that in $\mathcal{B}_{ab}(n)$. Together with compositionality and associativity of parallel composition, that is, $(P \parallel Q) \parallel R \approx_L P \parallel (Q \parallel R)$, this allows us to conclude that $SWP \approx_{L_1} B^{n+2}$.

Evaluation The proof-script with the bisimulations verifying the SWP contains about 600 lines, and has been set up in less than two weeks. The proofs of (2) and (3) are rather straightforward, as the processes related by the bisimulations behave in similar ways. Isabelle therefore deduces the proofs automatically without the user having to split them into single theorems covering the obligations. Note that the weak transitions are directly derivable from strong ones, thus need not be given by the user. Also, almost no additional results about the lists have to be provided by the user. The most challenging part concerning the mechanization was the proof of the first part of (4), as here the bisimulation maps the cyclic lists of B^P to the linear lists stored in $B^{n+2}(\Box^n)$. The corresponding theorems make up for nearly two thirds of the proof; their derivation necessitates certain expertise in theorem proving.

5.3 Up-To Techniques

When setting up new results in a theorem-prover, one is usually interested in splitting large arguments into smaller sub-proofs. The more often such a sub-argument can be reused, the better. In Section 5.1, for instance, we have split bisimulation proofs into single proofs for diverse bisimulation obligations. In Section 5.2, we have exploited compositionality of observation equivalence. In this section, we discuss the application of 'up-to' techniques that have originally been introduced by Milner in [Mil89], and have been further developed as a powerful proof-technique by Sangiorgi and Milner in [SM92]. Up-to techniques are particularly interesting, because they aim at reducing the size of a bisimulation relation by transferring parts of the behaviour of the contained systems to auxiliary bisimulation proofs. That is, instead of showing that a relation $\mathcal{R}$ is a bisimulation, one chooses some suitable $\mathcal{S}$ and proves that $\mathcal{S} \circ \mathcal{R} \circ \mathcal{S}^{-1}$ is a bisimulation. Observe that the sizes of of $\mathcal{S}$ and $\mathcal{R}$ can be by far smaller than that of $\mathcal{S} \circ \mathcal{R} \circ \mathcal{S}^{-1}$. The theory of 'up-to' techniques—in particular, how a suitable $\mathcal{S}$ should look like—has been further investigated by Sangiorgi [San95]. In this section, we are concerned with the application in theorem proving of one such technique, called 'up-to-expansion'.

Intuitively, an expansion is a bisimulation in which the first process performs at least as many internal steps as, or is less efficient than, the second one. Expansion can be used to abstract, for instance, from internal timers, or from internal communication of data. Every expansion (wrt. L) is a bisimulation (wrt. L). The converse does not hold.

Definition 5.3 (Expansion wrt. L) A relation $\mathcal{R} \subseteq \mathcal{P} \times \mathcal{P}$ is an *expansion wrt.* $L \subseteq \mathcal{N}$, if for all $P\mathcal{R}Q$, all $\alpha \in \mathcal{N} \cup \overline{\mathcal{N}} \cup \{\tau\}$, and all $P', Q' \in \mathcal{P}$,

(i) if $P \stackrel{\alpha}{\longrightarrow} P'$, there exist P' such that $Q \stackrel{\hat{\alpha}}{\longrightarrow} Q'$ and $P'\mathcal{R}Q'$.

(ii) if $Q \stackrel{\alpha}{\longrightarrow} Q'$, there exists a P' such that $P \stackrel{\alpha}{\Longrightarrow} P'$ and $P'\mathcal{R}Q'$.

Expansion wrt. L is the union $\geq_L \stackrel{\text{def}}{=} \bigcup\{\, \mathcal{R} \mid \mathcal{R}$ is a weak bisimulation wrt. $L\,\}$.

Definition 5.4 (Bisimulation up to Expansion) *A relation* $\mathcal{R} \subseteq \mathcal{P}\times\mathcal{P}$ *is a* bisimulation up to expansion wrt. $L \subseteq \mathcal{N}$ *if for all* $P\mathcal{R}Q$*, all* $\alpha \in L \cup \{\tau\}$*, all* $P' \in \mathcal{P}$*, and all* $Q' \in \mathcal{P}$*,*

(i) If $P \stackrel{\alpha}{\longrightarrow} P'$*, there exist* $P'', Q', Q'' \in \mathcal{P}$ *such that* $Q \stackrel{\hat{\alpha}}{\Longrightarrow} Q'$ *and* $P' \leq_L P''\mathcal{R}Q'' \geq_L Q'$.

(ii) If $Q \stackrel{\alpha}{\longrightarrow} Q'$*, there exists* $P', P'', Q'' \in \mathcal{P}$ *such that* $P \stackrel{\hat{\alpha}}{\Longrightarrow} P'$ *and* $P' \leq_L P''\mathcal{R}Q'' \geq_L Q'$.

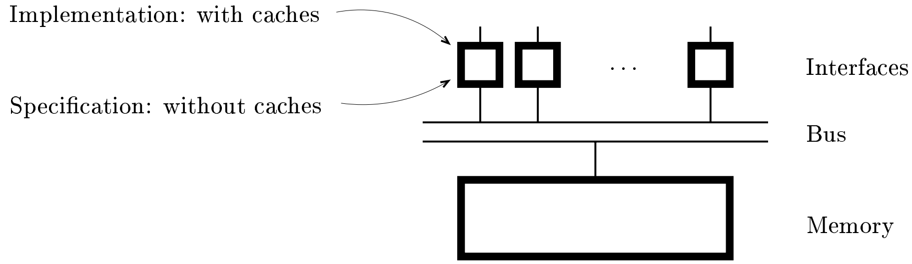

Figure 5.3: The structure of implementation and specification. In order to read from or write to the memory, processors have to connect to one of the interfaces. The interfaces of the implementation possess caches, whereas those of the specification have to pass every read request to the main memory.

It is a standard exercise to show that two states are observationally equivalent (wrt. L) if they are bisimilar up to expansion (wrt. L). It can be established by proving that for every bisimulation $\mathcal{R}$ (wrt. L), the relation $\approx_L \mathcal{R} \approx_L$ is a bisimulation wrt. L; the proof consists of simple diagram-chasing.

Proposition 5.5 *If $\mathcal{R} \subseteq \mathcal{P} \times \mathcal{P}$ is a bisimulation up to expansion wrt. $L \subseteq \mathcal{N}$, then $\approx_L \mathcal{R} \approx_L$ is a bisimulation wrt. L.*

Proof: The proof is obvious using standard diagram-chasing. □

5.3.1 Write-Invalidate Cache-Coherence

As a last case study, we have chosen an example which is parameterized both in number and size of its components. An interesting point is that internal communications follow a broadcast-mechanism. In our formalism, this can be easily taken care of by modifying the introduction rules for parallel composition. The new rules are,

- $P_i \xrightarrow{\alpha} P_i'$ implies $P_1 \parallel \ldots \parallel P_i \parallel \ldots \parallel P_n \xrightarrow{\alpha} P_1 \parallel \ldots \parallel P_i' \parallel \ldots \parallel P_n$;
- $P_i \xrightarrow{\bar{a}(v)} P_i'$ and $\forall j \neq i.\ P_j \xrightarrow{a(v)} P_j'$ imply $P_1 \parallel \ldots \parallel P_n \xrightarrow{\tau} P_1' \parallel \ldots \parallel P_n'$.

An interesting point of this case-study is that implementation and specification have the same topology; that is, from an observer's point of view they are built from the same components, yet a significant part of the components behave differently.

We verify the coherence of a simple write-invalidate cache-protocol, as described, for instance, in [HP96]; our case-study was motivated by a similar verification of the protocol in ACL2 by Moore [Moo98]. There, however, coherence is stated as a first-order property, whereas following our approach it can be stated as a corollary of a much stronger result: we show that an application of this write-invalidate cache-protocol yields a system that cannot be distinguished from a 'classical' memory without caching. Like Moore, we give a parameterized proof, so that our result is applicable to systems of arbitrary size.

	The memory, used by specification and system		
M1	$M(s) \stackrel{\mathrm{rq}_r(i)}{\longrightarrow} M(s,i)$		(lookup)
M2	$M(s,i) \stackrel{\overline{\mathrm{ret}}(s[i])}{\longrightarrow} M(s)$		(return)
M3	$M(s) \stackrel{\mathrm{rq}_w(i,v)}{\longrightarrow} M(s\{v/s[i]\})$		(update)
	Specification: interfaces to memory without caches		
I1	$I_l(\mathrm{stat},\mathrm{free}) \stackrel{\mathrm{read}_l(i)}{\longrightarrow} I_l(\mathrm{stat},\mathrm{read}(i))$	if $i \le \lvert s \rvert$	(read)
I2	$I_l(\mathrm{awake},\mathrm{read}(i)) \stackrel{\overline{\mathrm{rq}}_r(i)}{\longrightarrow} I_l(\mathrm{awake},\mathrm{wait})$		(lookup)
I3	$I_l(\mathrm{awake},\mathrm{wait}) \stackrel{\mathrm{ret}(v)}{\longrightarrow} I_l(\mathrm{awake},\mathrm{del}(v))$		(return)
I4	$I_l(\mathrm{stat},\mathrm{del}(v)) \stackrel{\overline{\mathrm{return}_l}(v)}{\longrightarrow} I_l(\mathrm{stat},\mathrm{free})$		(serve)
I5	$I_l(\mathrm{stat},\mathrm{free}) \stackrel{\mathrm{write}_l(i,v)}{\longrightarrow} I_l(\mathrm{stat},\mathrm{write}(i,v))$	if $i \le \lvert s \rvert$	(write)
I6	$I_l(\mathrm{awake},\mathrm{write}(i,v)) \stackrel{\overline{\mathrm{rq}}_w(i,v)}{\longrightarrow} I_l(\mathrm{awake},\mathrm{free})$		(update)
I7	$I_l(\mathrm{awake},\mathrm{req}_l) \stackrel{\mathrm{rq}_r(i)}{\longrightarrow} I_l(\mathrm{asleep},\mathrm{req}_l)$		(sleep)
I8	$I_l(\mathrm{asleep},\mathrm{req}_l) \stackrel{\mathrm{ret}(v)}{\longrightarrow} I_l(\mathrm{awake},\mathrm{req}_l)$		(wake up)
I9	$I_l(\mathrm{awake},\mathrm{req}_l) \stackrel{\mathrm{rq}_w(i,v)}{\longrightarrow} I_l(\mathrm{awake},\mathrm{req}_l)$		(ignore)
	Implementation: interface to memory with caches		
C2a	$I^c_l(c_l,\mathrm{stat},\mathrm{read}(i)) \stackrel{\tau}{\longrightarrow} I^c_l(c_l,\mathrm{stat},\mathrm{wait})$	if valid($c_l[i]$)	(cache)
C2b	$I^c_l(c_l,\mathrm{awake},\mathrm{read}(i)) \stackrel{\overline{\mathrm{rq}}_r(i)}{\longrightarrow} I^c_l(c_l,\mathrm{awake},\mathrm{wait})$	if invalid($c_l[i]$)	(lookup)
C6	$I^c_l(c_l,\mathrm{awake},\mathrm{write}(i,v)) \stackrel{\overline{\mathrm{rq}}_w(i,v)}{\longrightarrow} I^c_l(c_l\{(v,\mathrm{valid})/i\},\mathrm{awake},\mathrm{free})$		(update)
C9	$I^c_l(c_l,\mathrm{awake},\mathrm{req}_l) \stackrel{\mathrm{rq}_w(i,v)}{\longrightarrow} I^c_l(c_l\{(c_l[i],\mathrm{invalid})/i\},\mathrm{awake},\mathrm{req}_l)$		(invalidate)

Table 5.6: **Write-invalidate cache-coherence.** Rules **M1** to **M3** describe the behaviour of the memory. Rules **I1** through **I9** specify the behaviour of each interface. The behaviour of the implementation is given by rules **C1** through **C9** similar to **I1** to **I9** except for the rules that are explicitely shown above, which involve cache cells. Rules **I1**–**I4**, respectively **C1**–**C4**, describe the behaviour of interfaces I_l and I^c_l in the presence of a read request. **I5**, **I6** and **C5**, **C6** handle write requests. **I7**, **I8** and **C7**, **C8** describe how the interfaces fall asleep whenever they encounter a read request on the bus. **I9** and **C9** concern the detection that some other interface writes on the memory.

The system: implementation and specification Figure 5.3 shows a graphical representation of the topology of the system, both for implementation and specification. We assume a distributed computer network with a monolithic *main memory* M. Note that we do not make any assumptions about the actual structure of the main memory; we simply model it in terms of a component. This means that any memory which is observation-equivalent to M could be used, owing to compositionality as discussed in Section 5.2. Communication with the main memory takes place over a *bus*. We do not model this bus as a process of its own, but use a synchronous broadcast mechanism to describe it. Computers can connect to the bus via *interfaces* I_i^c (in the implementation) and I_i (in the specification), which are again modelled in terms of components. The difference between an I_i^c and an I_i is that I_i^c possesses a *cache* memory—so that it need not necessarily communicate with M upon a request—whereas I_i does not possess one—and hence has to access M upon every request.

The memory M is modelled in terms of a component, as described by rules **M1**–**M3** in Table 5.6. Upon a read-request from one of the interfaces (**M1**), it simply returns the desired value (**M2**), and upon a write-request, it updates the specified cell without returning an acknowledgement (**M3**). Observe that none of the communications along the bus specifies the index (or, address) of the calling interface; this is not necessary, because whenever an interface sends a read-request, this will be noticed by the others so that they can await the answer before making requests themselves.

Before explaining the behaviour of the I_i^c-components, we describe that of the I_i-components, because it is simpler and can be used as a common basis for the description of how an I_i^c works. Rules **I1**–**I9** in Table 5.6 formally define the operational semantics of the I_i. An interface I_i that is currently free can accept read- or write-requests from the environment. Upon a read-request (**I1**), it asks the main memory to return the desired value (**I2**), and upon its receipt (**I3**) returns it to the environment (**I4**). Upon a write-request (**I5**), it tells the main memory to update its respective cell accordingly (**I6**). Observe that an interface can either be *active* (or, *awake*) or *passive* (or, *asleep*), and that only active interfaces can submit read- or write-requests. The reason is that whenever an interface performs a read-request, all others have to wait for the request to be served before submitting themselves; for the time being, they fall asleep (**I7**, **I8**). We exploit the underlying broadcast-mechanism to model that all interfaces continuously snoop on the bus for requests. Due to there being no answer, write-requests are ignored (**I9**).

The behaviour of the I_i^c-components is similar, except that some of the read-requests can be served from the cache, and that the caches have to be updated upon write-requests. Table 5.6 therefore only presents those rules that are different form **I1**–**I9**. Upon a read-request (**C1**, corresponding to **I1**), I_i^c checks whether the addressed cell is valid in its cache; if it is, it takes the value from there (**C2a**); if it is not, it sends a request to the main memory (**C2b**). Upon a write-request (**C5**, corresponding to **I5**), it tells the main memory to update its respective cell accordingly, and simultaneously updates the respective cell in its cache, (re-)*validating* it (**C6**). This time, write-requests cannot be ignored by the other interfaces; instead, they *invalidate* their corresponding cache-cells (**C9**). Read-requests, on the other hand, are treated in **C7** and **C8** like in **I7** and **I8**: whenever an interface encounters one, it falls asleep until the request has been served.

The visible behaviour of the system, that is, both of implementation and specification, consists of accepting read- and write-requests from the environment, and answering the read-

requests. It can be described by a set L_3, defined as follows:

$$L_3 \stackrel{\text{def}}{=} \{\text{read}_l(i),\ \text{write}_l(i,v),\ \text{return}_l(v) \mid l, i \in I\!N,\ v \text{ of suitable type}\}.$$

With this, we can show that implementation and specification behave in an equivalent way,

$$M(s) \parallel I_1^c(t_1) \parallel \cdots \parallel I_n^c(t_n) \quad \approx_{L_3} \quad M(s) \parallel I_1(u_1) \parallel \cdots \parallel I_n(u_n),$$

for every content s of the memory M, and for all possible tuples of parameters $t_1, \ldots, t_n$ and $u_1, \ldots u_n$, where $t_l = (c_l, \text{stat}_l, \text{req}_l)$ and $u_l = (\text{stat}_l, \text{req}_l)$, and valid$(c_l[i])$ implies $c_l[i] = s[i]$, for all i. Note that this last condition reflects the safety-property of the protocol, which is guaranteed by observation equivalence.

Remark: Observe that a value returned to the environment as the value of some cell by an interface need not necessarily coincide with the current value of that cell, neither in the implementation nor in the specification. The reason is that an interface may delay the delivery of a value it has already retrieved from the memory or from its cache, even until the cell has been updated in the memory by another interface.

Bisimulation up-to expansion We exhibit a family of bisimulations up to expansion, $\mathcal{R}_{n,m}$ that are parameterized in the number of interfaces, n, and in the size of the store, m. The size of each of these relations depends on the size of the type of the data stored in the memory; if it is infinite, $\mathcal{R}_{n,m}$ is infinite as well. This example therefore bears three potential sources of infinity: (1) infinity of the data-type, (2) infinity due to a parameterized list describing the store, and (3) infinity due to a parameterized number of components.

For fixed but arbitrary n and m, the relation $\mathcal{R}_{n,m}$ contains pairs of states of the implementation and the specification in which all the interfaces are awake, that is, are allowed to access the bus immediately, all caches are coherent, and in which the requests in the I_l^c and I_l are identical for all l,

$$\begin{aligned}\mathcal{R}_{n,m} \stackrel{\text{def}}{=} \{\ &(M(s) \parallel I_1^c(c_1, \text{awake}, \text{req}_1) \parallel \ldots \parallel I_n^c(c_n, \text{awake}, \text{req}_n),\\ &\ M(s) \parallel I_1(\text{awake}, \text{req}_1) \qquad \parallel \ldots \parallel I_n(\text{awake}, \text{req}_n)) \quad \mid \\ &|s| = m \ \wedge\ \forall\, 1 \le l \le n,\ 1 \le i \le m.\ \ |c_l| = m \ \wedge\ \text{valid}(c_l[i]) \Rightarrow c_l[i] = s[i]\ \}.\end{aligned}$$

The relation $\mathcal{R}_{n,m}$ is a bisimulation up to expansion wrt. L_3. It contains pairs of systems in which all interfaces are active, valid cache cells are coherent with the main memory, and corresponding I_i^c and I_i carry corresponding requests. Of course, $\mathcal{R}_{n,m}$ does not describe the entire behaviour of the systems; for instance, it does not consider states in which interfaces are asleep, although these are obviously reachable. These states are dealt with by auxiliary expansion relations $\mathcal{E}_{n,m}^c \cup \mathcal{I}d_{\mathcal{P}}$ for the implementation, and $\mathcal{E}_{n,m} \cup \mathcal{I}d_{\mathcal{P}}$ for the specification, where $\mathcal{I}d_{\mathcal{P}}$ denotes the identity relation for processes. These expansions relate states directly before and after a read-request has been served by the main memory; these states are observationally

indistinguishable.

$$
\begin{array}{rll}
\mathcal{E}^c_{n,m} \stackrel{\text{def}}{=} \{ & (M_r(s,i) \parallel I^c_1(c_1, \text{asleep}, \text{req}_1) \parallel \ldots \parallel & \\
& \quad I^c_l(c_l, \text{awake}, \text{wait}) \parallel \ldots \parallel I^c_n(c_n, \text{asleep}, \text{req}_n), & \\
& M(s) \quad \parallel I^c_1(c_1, \text{awake}, \text{req}_1) \parallel \ldots \parallel & \\
& \quad I^c_l(c_l, \text{awake}, \text{del}(s[i])) \parallel \ldots \parallel I^c_n(c_n, \text{awake}, \text{req}_n)) & \mid \\
& |s| = m \quad \wedge \quad \forall\, 1 \le l \le n,\ 1 \le i \le m. \ \ |c_l| = m \ \wedge \ \text{valid}(c_l[i]) \Rightarrow c_l[i] = s[i] \,\} &
\end{array}
$$

$$
\begin{array}{rll}
\mathcal{E}_{n,m} \stackrel{\text{def}}{=} \{ & (M_r(s,i) \parallel I_1(\text{asleep}, \text{req}_1) \parallel \ldots \parallel & \\
& \quad I_l(\text{awake}, \text{wait}) \parallel \ldots \parallel I_n(\text{asleep}, \text{req}_n), & \\
& M(s) \quad \parallel I_1(\text{awake}, \text{req}_1) \parallel \ldots \parallel & \\
& \quad I_l(\text{awake}, \text{del}(s[i])) \parallel \ldots \parallel I_n(\text{awake}, \text{req}_n)) & \mid \\
& |s| = m \,\} &
\end{array}
$$

The whole proof then falls into the following two steps: (1) prove that $\mathcal{I}d_{\mathcal{P}}$, every $\mathcal{E}^c_{n,m} \cup \mathcal{I}d_{\mathcal{P}}$, and every $\mathcal{E}_{n,m} \cup \mathcal{I}d_{\mathcal{P}}$ are expansions wrt. L_3; and, (2) show that every $\mathcal{R}_{n,m}$ is a bisimulation up to expansion wrt. L_3, exploiting the results provided in step (1). The proofs follow the same pattern as those in Section 5.1. We sketch part (2). After applying Definition 5.4 as a rewrite-rule and the case-exhaustion generated from the transition-rules, Isabelle/HOL automatically computes seven proof-obligations—four for I^c_l and three for I_l for some fixed but arbitrary interface l—of which we consider two typical cases:

1. If I^c_l sends a read-request along the bus, and all other I^c_k fall asleep, also I_l should send a read-request with all I_k falling asleep. By expansion, these states should then yield another pair of states in $\mathcal{R}_{n,m}$, where the read-requests have already been served. This is easy to show in the prover, closely following the definition of $\mathcal{R}_{n,m}$, and applying the expansion results from $\mathcal{E}^c_{n,m}$ and $\mathcal{E}_{n,m}$. Like in almost all cases, a strong transition can be matched simply by a strong transition, hence not much interaction from the user is necessary.

2. If I^c_l retrieves a value from its cache, I_l has to perform a corresponding chain of transitions. This proof-obligation requires interaction from the user: we choose that I_l should submit a read-request along the bus; afterwards we apply $\mathcal{I}d_{\mathcal{P}}$ as an expansion to the implementation, and $\mathcal{E}_{n,m}$ to the specification. The values to be delivered after this transition are shown to be equal in implementation and specification by applying the coherence-criterion from $\mathcal{R}_{n,m}$.

Evaluation The Isabelle/HOL proof-script consists of approximately 1000 lines of code. We were able to increase the profit from Isabelle's automatic tactics by splitting the proof obligations into separate subgoals. We further benefitted from the uniform structure of the single proofs, so we were able to reuse the same proof-procedure in several obligations.

Standard bisimulation In a standard bisimulation proof, a lot more combinations have to be considered; both systems can be asleep, for instance, or one system is asleep waiting for a read-request to be served while the other is awake. This yields composite bisimulation-relations $\mathcal{R}^{st}_{n,m}$ wrt. L_3, where $\mathcal{R}^{st}_{n,m} \stackrel{\text{def}}{=} \mathcal{R}_{n,m} \cup \mathcal{R}^{ws}_{n,m} \cup \mathcal{R}^{ss}_{n,m}$. The relation $\mathcal{R}^{ws}_{n,m}$ contains those pairs of

states in which the implementation is awake and the specification is asleep,

$$\begin{array}{lll}\mathcal{R}^{ws}_{n,m} & \stackrel{\text{def}}{=} & \{\ (M(s) \parallel I^c_1(c_1, \text{awake}, \text{req}_1) \parallel \ldots \parallel I^c_l(c_l, \text{awake}, \text{del}(s[i])) \\ & & \qquad\qquad \parallel \ldots \parallel I^c_n(c_n, \text{awake}, \text{req}_n), \\ & & M_r(s,i) \parallel I_1(\text{asleep}, \text{req}_1) \parallel \ldots \parallel I_l(\text{awake}, \text{wait}) \\ & & \qquad\qquad \parallel \ldots \parallel I_n(\text{asleep}, \text{req}_n)) \quad | \\ & & |s| = m \quad \wedge \quad \forall 1 \leq l \leq n,\, 1 \leq i \leq m. \ \ |c_l| = m \ \wedge\ \text{valid}(c_l[i]) \Rightarrow c_l[i] = s[i]\ \}\end{array}$$

The relation $\mathcal{R}^{ss}_{n,m}$ contains those in which both are asleep,

$$\begin{array}{lll}\mathcal{R}^{ss}_{n,m} & \stackrel{\text{def}}{=} & \{\ (M_r(s,i) \parallel I^c_1(c_1, \text{asleep}, \text{req}_1) \parallel \ldots \parallel I^c_l(c_l, \text{awake}, \text{wait}) \\ & & \qquad\qquad \parallel \ldots \parallel I^c_n(c_n, \text{asleep}, \text{req}_n), \\ & & M_r(s,i) \parallel I_1(\text{asleep}, \text{req}_1) \parallel \ldots \parallel I_l(\text{awake}, \text{wait}) \\ & & \qquad\qquad \parallel \ldots \parallel I_n(\text{asleep}, \text{req}_n)) \quad | \\ & & |s| = m \quad \wedge \quad \forall 1 \leq l \leq n,\, 1 \leq i \leq m. \ \ |c_l| = m \ \wedge\ \text{valid}(c_l[i]) \Rightarrow c_l[i] = s[i]\ \}\end{array}$$

Pairs in $\mathcal{R}^{ws}_{n,m}$ are due to an I^c_l retrieving a value from its cache while the corresponding I_l has to look it up in the main memory. The combination $\mathcal{R}^{sw}_{n,m}$, in which the interfaces of the implementation are asleep while those of the specification are not, need not be considered: whenever an I^c_l sends a read request along the bus, there is no need for the corresponding I_l to go a step further and serve the request.

Evaluation In order to reduce the number of cases of this proof, we have modified the transition systems so that interfaces that are asleep cannot even communicate with the environment. But still the proof script is as long as that using bisimulations up to expansion, that is, around 1000 lines. Without the simplification, it would have been even longer.

5.4 Discussion

In this chapter, we have described how to obtain a general validation-technique for systems with an infinite number of states, by applying the proof-method of observation-equivalence in interactive theorem-proving, and have examined it on various benchmark examples. We now present conclusions drawn from these case-studies, discussing both advantages and deficiencies of the approach, and compare our approach to related work in the field.

Range of applicability Communication protocols, as presented in Sections 5.1 and 5.2, are often intended to behave like variations of buffers: messages are to be transmitted in the order in which they arrive. This means that the specification of a communication protocol usually can be described in terms of an abstract system, so that observation equivalence is a natural specification technique. Also in hardware verification, as seen in Section 5.3, the specification usually can be described in terms of a system itself, and can be made subject to observation equivalence. A less natural field of application of process equivalences in general is the area of distributed algorithms, where specifications often consist of lists of properties a system should satisfy. For some of them, one can consider the most liberal system satisfying the properties,

and prove that the implementation behaves "as well as, or better than" this system. For this purpose, observation equivalence is replaced by the observation-preorder.

In general, the technique is most suitable for systems that can be concisely described but have infinitely many states and use some nontrivial datatypes. These systems are still out of reach for fully automatic tools, but lead to manageable bisimulations.

Is observation equivalence suitable? Observation equivalence has been argued to be too discriminating in practice; in fact, often language (or, trace) equivalence [Mil89] is preferred, as, for instance, in [PS91]. In the area of communication protocols, this question seems to be a lesser problem. Many specifications are deterministic, and in this case, observation and fair testing equivalence [NC95]—and sometimes even language equivalence—coincide. Compared to these two equivalences, observation equivalences offers a better proof methodology. Thus, in cases where the equivalences coincide, one can profit from bisimulation proof techniques in order to show language or testing equivalence.

Sometimes one might be interested in over-specifications. In these cases, *observation preorder* offers proof techniques based on *simulations*, considering only one direction of a bisimulation, see [Mil89] for a formal definition. The resulting proof methodology is similar, consisting of one half of the definition of bisimulations; and compositionality can be exploited as well.

Keeping bisimulations manageable Keeping the size of relations manageable is an important problem of our approach. Compositionality of observation equivalence is a big help, as we could see in Section 5.2: if we had not been able to replace the ABP channels by one-place buffers, the bisimulation would have been unmanageable. Furthermore, there exist various up-to techniques, one of which is bisimulation up to expansion, see Section 5.3.

How to find a bisimulation? Searching for a bisimulation is an incremental process. Usually, one starts with some base state of implementation and specification, and adds pairs, or (probably infinite) families of pairs, until one has obtained a bisimulation. This approach can be formally described in terms of the *coinductive* method of fixed-point generation (see, for instance, [Rut98]), and is supported by Isabelle/HOL [Pau93]. The advantage of coinduction is that finding the relation and proving that it is a bisimulation are intertwined, and Isabelle deals with all technical details as, for instance, "has pair (s, t) already been considered or not?" One inconvenience is, however, that the bisimulation cannot yet be extracted as an Isabelle constant from the coinductive proof, to be available for further use, and has to be added by hand.

Dealing with data structures Proving simple facts about the data structures of a system (list, stacks, et cetera) may amount to more than half of the interaction with the theorem prover. These facts are stored in Isabelle's database for future use, but their application still requires considerable expertise in theorem proving. The user may decide not to perform the full proof, by taking theorems about data structures as unproved axioms. For simple theorems this is a sensible approach, since the proof loses almost no credibility.

What about wrong implementations? Assume one tries to verify that the implementation I of a system matches its specification S, although it does not. This is the 'hard' case using observation equivalence: no matter how hard one tries, one will always end up with a pair

of states that do not match. The question is: Can the search for a bisimulation be exploited methodically, so that inequivalence can be proved? And how can inequivalence be proved? Often, an incorrect implementation even produces a language that is different from that of the specification. Keeping this in mind, one can backtrack the coinductive path of building up a bisimulation-relation up to the point where I and S produce different transitions. Like this, a distinguishing trace has been constructed. Here, the theorem-prover can be used as a book-keeper; previous steps can be reproduced (manually, however) from the proof-script.

General evaluation and future work The approach is certainly labour intensive when compared to automatic verification. It is useful for infinite-state systems with not too large a description which do not exhibit regularity properties making it amenable to model checking. The approach is particularly suitable for modular systems in which each of the modules has a separate specification. Future work should concentrate in the interactive design of the bisimulation. As mentioned above, coinduction-techniques for this problem are available in Isabelle, but they are still very unfriendly to the user.

Chapter 6

Conclusion and Future Work

This thesis has as its goal the outline of mechanized and mechanizable validation-techniques for infinite-state and higher-order concurrent systems, in the context of process-algebra. State-explosion and undecidability problems make the use of fully automatic techniques imposiible for the treatment of large and infinite-state systems. We therefore base tool-support on interactive theorem-proving. As a consequence, we have to select a human-style proof-technique. We choose observation-equivalence, because of its natural methodology—the obligations imposed by its definition are simple and uniformly applicable to diverse frameworks—and because it is obviously well-suited for a formalization. We apply bisimulations and bisimulation-based proof-techniques in the contexts of CCS, the π-calculus, and Concurrent Idealized Algol, and mechanize proofs for CCS-processes in Isabelle/HOL. The results obtained in this thesis can be regarded as the outline of an integrated framework for the validation of reactive and mobile systems in interactive theorem-provers. (1) We recommend that a framework be based on a simple but expressive language such as the π-calculus. Syntactic simplicity is essential with regard both to its formalization in a logical framework and to the effectivity of proof-techniques. (2) Programs should be written in higher-level languages, nevertheless, and then translated into the description-language so that they can be verified operationally within the theorem-prover. (3) For the validation, we choose bisimulation-based proof-techniques. In order to validate equivalences of higher-order programs, we employ bisimulations up to expansion and contextual reasoning. These techniques can be further enhanced by adding algebraic techniques and a coinductive construction of bisimulations. In the remainder of this chapter, we discuss the contributions of this thesis to each of the three questions, and sketch possible directions for future work.

(1) In Chapter 3, we have presented a shallow embedding of the monadic π-calculus in Isabelle/HOL. By introducing well-formedness predicates for processes and process-abstractions, we have eliminated exotic terms and, simultaneously, obtained principles for syntax-induction. With these, we have been able to derive vital syntactic properties of the π-calculus and prove that the encoding is adequate, both fully within Isabelle/HOL.

The framework can now be used for a semantic analysis of the π-calculus in HOAS. It can serve as a basis for the comparison of different semantics for the π-calculus and as a platform for the analysis of processes modelled in the π-calculus. Two semantics for the π-calculus have been studied in shallow embeddings in Coq [HMS00, Des00]. Both of them do not immediately instantiate continuations of input-processes, hence do not need to reabstract over once instantiated names. While the first semantics follows a classical late approach, the second

uses abstractions and concretions. It is an open question which of these and other semantic models applies most naturally in a shallow embedding. Also, it will have to be investigated whether and how the theory of contexts has to be modified to deal with models like that proposed in [Des00].

A recent strand of research investigates structural-induction principles for higher-order logical frameworks such as Elf or its successor Twelf [Pfe89, DPS97, PS99]. Once this has been established, it will be interesting to see how the syntactic principles we have derived in Chapter 3 can be proved there. Adequacy will not be provable within these systems, however, because they do not offer means for first-order descriptions. On the other hand, adequacy is guaranteed by the outline of these frameworks. Further, they might apply better in semantic analysis, because no explicit well-formedness predicates have to be considered by the user.

(2) In Chapter 4, we have designed a sound π-calculus semantics for Concurrent Idealized Algol (CIA), and have used it to prove classical laws, as well as a more complex example involving procedures of higher order. Our choice of CIA is especially motivated by the fact that it has a small syntax but can nevertheless be considered as a paragon of concurrent imperative higher-order programming-languages. Up to this point, we do not see any means for finding suitable behavioural equivalences that work directly on the operational semantics of CIA. Our work is therefore in line with game-semantics for sequential IA [AM96a, AM99].

When modelling 'real-life' programming-languages in the π-calculus, mechanized support for proofs will be indispensable on the long run. It could build on formalizations such as the one given in Chapter 3 of this thesis, augmented with a compiler to translate terms from the original language into the π-calculus. A very interesting, though rather theoretically oriented, project is the mechanization of proofs in the style of Chapter 3 in an interactive theorem-prover, no matter for which language. A proof of this size could turn out to be a hard test for logical frameworks, exploiting induction of various kinds, and several formulations of bisimulation-equivalences.

More recently, the design of concurrent programming-languages has centered on an extension of higher-order process-calculi with conveniences for the development of larger software-systems, such as modularization or exception-handling [Tur95, FMS00, Ode00]. Proofs about programs written in these languages can then be conducted within the underlying process-calculi. Theorem-prover support could be helpful here as well.

(3) In Chapter 5, we have demonstrated by larger examples from hardware-verification how to check bisimulations in interaction with a general-purpose theorem-prover. In particular, we have demonstrated the applicability of this kind of mechanization to infinite-state and parameterized systems, which cannot be verified fully automatically. The outlined verification-technique is applicable in various ways: because it uses normal-forms, it can be adapted for mobile systems in a straightforward way. Further, it can be used in a combination of theorem-proving and model-checking, by proving that an infinite-state (or, large) system is bisimilar to a finite-state (or, smaller) specification, which can then be subject to a model-checking procedure.

Still, the user has to set up the bisimulation on his/her own. Here, a further development of interactive coinductive techniques might be useful. Further, when applying the proposed proof-methodology in large style, the automatic tactics have to be enhanced by search-strategies that are able not only to deal with the most trivial cases. Further, an integrated framework should consider a range of methods, including up-to techniques as well as algebraic reasoning. This

is particularly important with regard to the validation of programs written in higher-level languages like CIA.

Bibliography

[AKH92] S. Arun-Kumar and M. Hennessy, *An efficiency preorder for processes*, Acta Informatica **29** (1992), 737–760.

[AM96a] S. Abramsky and G. McCusker, *Linearity, sharing and state: a fully abstract game semantics for Idealized Algol with active expressions*, Proc. Linear Logic meeting'96, ENTCS, vol. 3, Elsevier, 1996.

[AM96b] O. Ait-Mohamed, *Pi-calculus theory in HOL*, Ph.D. thesis, Henry Poincarré University, Nancy, 1996.

[AM99] S. Abramsky and G. McCusker, *Full abstraction for Idealized Algol with passive expressions*, Theoretical Computer Science **227** (1999), no. 1, 3–42.

[Ama93] R. Amadio, *On the reduction of CHOCS bisimulation to π-calculus bisimulation*, Proc. CONCUR'93, LNCS, vol. 715, Springer, 1993, pp. 112–126.

[Bar81] H. Barendregt, *The lambda-calculus, its syntax and semantics*, North-Holland, 1981.

[BBC+99] B. Barras, S. Boutin, C. Cornes, J. Courant, Y. Coscoy, D. Delahaye, D. de Rauglaudre, J. Filliâtre, E. Giménez, H. Herbelin, G. Huet, H. Laulhère, C. Muñoz, C. Murthy, C. Parent-Vigouroux, P. Loiseleur, C. Paulin-Mohring, A. Saïbi, and B. Werner, *The Coq proof assistant reference manual – version 6.3.1*, Tech. report, INRIA, 1999.

[BG88] M. Bezem and J. Groote, *A formal verification of the alternating bit protool in the calculus of constructions*, Tech. report, Utrecht University, 1988.

[BKM96] B. Brock, M. Kaufmann, and J. Moore, *ACL2 theorems about commercial microprocessors*, Proc. FMCAD'96, LNCS, vol. 1166, Springer, 1996, pp. 275–293.

[Bro96] S. Brookes, *The essence of parallel Algol*, Proc. LICS'96, IEEE Press, 1996, App. in vol. 2 of [OT97], pp. 164–173.

[BSW69] K. A. Bartlett, R. A. Scantlebury, and P. T. Wilkinson, *A note on reliable full-duplex transmission over half-duplex links*, Communications of the ACM **12** (1969), no. 5, 260–261.

[BW90] J. Baeten and W. Weijland, *Process algebra*, Cambridge University Press, 1990.

[BW99] S. Berghofer and M. Wenzel, *Inductive datatypes in HOL—lessons learned in Formal-Logic Engineering*, Proc. TPHOL'99, LNCS, vol. 1690, 1999, pp. 19–36.

[Chu40] A. Church, *A formulation of simple type theory*, Journal of Symbolic Logic **5** (1940), 56–68.

[deB72] N. deBruijn, *Lambda calculus notation with nameless dummies: a tool for automatic formula manipulation, with application to the Church-Rosser theorem*, Indagationes Mathematicae **34** (1972), 381–392.

[Des00] J. Despeyroux, *A higher-order specification of the π-calculus*, Proc. TCS'00, LNCS, Springer, 2000, To appear.

[DFH95] J. Despeyroux, A. Felty, and A. Hirschowitz, *Higher-order abstract syntax in Coq*, Proc. TLCA'95, LNCS, vol. 902, Springer, 1995, pp. 124–138.

[DH94] J. Despeyroux and A. Hirschowitz, *Higher-order abstract syntax with induction in Coq*, Proc. LPAR'94, LNCS, vol. 822, Springer, 1994, pp. 159–173.

[DPS97] J. Despeyroux, F. Pfenning, and C. Schürmann, *Primitive recursion for higher-order abstract syntax*, Proc. TLCA'97, LNCS, vol. 1210, Springer, 1997, An extended version will appear in TCS., pp. 147–163.

[FHJ95] W. Ferreira, M. Hennessy, and A. Jeffrey, *A theory of weak bisimulation for core CML*, Tech. report, University of Sussex, 1995.

[FMS96] M. Fiore, E. Moggi, and D. Sangiorgi, *A fully-abstract model for the π-calculus*, Proc. LICS'96, IEEE Press, 1996, pp. 43–54.

[FMS00] C. Fournet, L. Maranget, and A. Schmitt, *The jocaml language beta release: Documentation and user's manual*, Tech. report, INRIA, 2000.

[Gay00] S. Gay, *A framework for the formalisation of pi-calculus type systems in Isabelle/HOL*, Tech. report, University of Glasgow, 2000.

[Gim96] E. Gimenez, *An application of co-inductive types in Coq: Verification of the alternating bit protocol*, Proc. TYPES'95, LNCS, vol. 1158, Springer, 1996, pp. 135–152.

[GM93] M. Gordon and T. Melham, *Introduction to hol: a theorem-proving environment for higher-order logic*, Cambridge University Press, 1993.

[GM96] A. Gordon and T. Melham, *Five axioms of alpha-conversion*, Proc. TPHOL'96, LNCS, vol. 1125, Springer, 1996, pp. 173–190.

[GS95] J. Groote and J. Springintveld, *Focus points and convergent process operators*, Logic Group Preprint Series 142, Utrecht University, 1995.

[Hen99] L. Henry-Gréard, *Proof of the subject reduction property for a pi-calculus in Coq*, Tech. Report RR-3698, INRIA, 1999.

[HHP93] R. Harper, F. Honsell, and G. Plotkin, *A framework for defining logics*, Journal of the Association of Computing Machinery **40** (1993), no. 1, 143–184.

[Hir97] D. Hirschkoff, *A full formalisation of π-calculus theory in the calculus of constructions*, Proc. TPHOL'97, LNCS, vol. 1275, Springer, 1997, pp. 153–169.

[HMS00] F. Honsell, M. Miculan, and I. Scagnetto, *π-calculus in (co)inductive type theory*, Theoretical Computer Science (2000), To appear.

[Hoa85] C. Hoare, *Communicating sequential processes*, Prentice-Hall, 1985.

[Hof99] M. Hofmann, *Semantical analysis of higher-order abstract syntax*, Proc. LICS'99, vol. 158, IEEE, 1999, pp. 204–213.

[Hon96] K. Honda, *Composing processes*, Proc. POPL'96, ACM Press, 1996, pp. 344–357.

[How96] D. Howe, *Proving congruence of bisimulation in functional programming languages*, Information and Computation **124** (1996), no. 2, 103–112.

[HP96] J. Hennessy and D. Paterson, *Computer architecture – a quantitative approach*, Morgan Kaufmann, 1996.

[Jon93] C. Jones, *A π-calculus semantics for an object-based design notation*, Proc. CONCUR'93, LNCS, vol. 715, Springer, 1993, pp. 158–172.

[KMM00a] M. Kaufmann, P. Manolios, and J. Moore, *Computer-aided reasoning: ACL2 case studies*, Kluwer Academic Publishers, 2000.

[KMM00b] M. Kaufmann, P. Manolios, and J. Moore, *Computer-aided reasoning: An approach*, Kluwer Academic Publishers, 2000.

[Kob97] N. Kobayashi, *A partially deadlock-free typed process calculus*, Proc. LICS'97, IEEE Press, 1997, pp. 128–139.

[KPT99] N. Kobayashi, B. Pierce, and D. Turner, *Linearity in the π-calculus*, Transactions on Programming Languages and Systems **21** (1999), no. 5, 914–947, A preliminary version appeared in Proc. POPL'96.

[KS98] J. Kleist and D. Sangiorgi, *Imperative objects and mobile processes*, Proc. PROCOMET'98, Chapman & Hall, 1998, pp. 285–303.

[Lyn68] W. C. Lynch, *Reliable full-duplex file transmission over half-duplex telephone lines*, Comm. of the ACM **11** (1968), no. 6, 407–410.

[Mam99] B. Mammass, *Méthodes et outils pour les preuve compositionnelles de systèmes paralleèles (in french)*, Ph.D. thesis, Pierre et Marie Curie University, Paris, 1999.

[Mel95] T. Melham, *A mechanized theory of the π-calculus in HOL*, Nordic Journal of Computing **1** (1995), no. 1, 50–76.

[Mil89] R. Milner, *Communication and concurrency*, Prentice-Hall, 1989.

[Mil91] R. Milner, *The polyadic π-calculus: A tutorial*, Tech. Report ECS–LFCS–91–180, Edinburgh University, 1991.

[Mil92a] D. Miller, *The π-calculus as a theory in linear logic: Preliminary results*, Proc. ELP'92, LNCS, vol. 660, Springer, 1992, pp. 242–264.

[Mil92b] R. Milner, *Functions as processes*, Journal of Mathematical Structures in Computer Science **17** (1992), 119–141.

[Mil99] R. Milner, *Communicating and mobile processes*, Cambridge University Press, 1999.

[Moo98] J. Moore, *An ACL2 proof of write invalidate cache coherence*, Proc. CAV'98, LNCS, vol. 1427, Springer, 1998, pp. 29–38.

[MP99] J. McKinna and R. Pollack, *Some lambda calculus and type theory formalized*, Journal of Automated Reasoning **23** (1999), no. 3–4, 373–409.

[MPW92] R. Milner, J. Parrow, and D. Walker, *A calculus of mobile processes*, Information and Computation **100** (1992), 1–77.

[MS88] A. Meyer and K. Sieber, *Towards fully abstract semantics for local variables*, Proc. POPL'88, 1988, App. in vol. 2 of [OT97], pp. 191–203.

[MT91] I. Mason and C. Talcott, *Equivalence in functional languages with effect*, Journal of Functional Programming **1** (1991), no. 3, 287–327.

[MT92] I. Mason and C. Talcott, *References, local variables and operational reasoning*, Proc. LICS'92, IEEE Press, 1992, pp. 186–197.

[NC95] V. Natarajan and R. Cleaveland, *Divergence and fair testing*, Proc. ICALP'95, LNCS, vol. 944, Springer, 1995, pp. 648–659.

[NM98] G. Nadathur and D. Miller, *An overview of λProlog*, Proc. LPC'98, MIT Press, 1998, pp. 810–827.

[NS94] T. Nipkow and K. Slind, *I/O automata in Isabelle/HOL*, Proc. TYPES'94, LNCS, vol. 996, Springer, 1994, pp. 101–119.

[Ode95] M. Odersky, *Polarized name passing*, Proc. FSTTCS, LNCS, vol. 1026, Springer, 1995, pp. 324–337.

[Ode00] M. Odersky, *Functional nets*, Proc. ESOP'2000, LNCS, vol. 1782, Springer, 2000, pp. 1–25.

[OT95] P. O'Hearn and R. Tennent, *Parametricity and local variables*, Journal of the ACM **42** (1995), no. 3, 658–709, App. in vol. 2 of [OT97].

[OT97] P. O'Hearn and R. Tennent (eds.), *ALGOL-like Languages*, Progress in Theoretical Computer Science, Birkhäuser, 1997, Two volumes.

[Par80] D. Park, *Concurrency and automata on infinite sequences*, LNCS, vol. 104, Springer, 1980.

[Pau93] L. Paulson, *Isabelle's object-logics*, Tech. Report 286, University of Cambridge, Computer Laboratory, 1993.

[Pau94] L. Paulson (ed.), *Isabelle: a generic theorem prover*, LNCS, vol. 828, Springer, 1994.

[Pfe89] F. Pfenning, *Elf: A language for logic definition and verified metaprogramming*, Proc. LICS'89, IEEE, 1989, pp. 313–321.

[Pit96] A. Pitts, *Reasoning about local variables with operationally-based logical relations*, Proc. LICS'96, IEEE Press, 1996, App. in vol. 2 of [OT97], pp. 152–163.

[PS91] K. Paliwoda and J. Sanders, *An incremental specification of the sliding-window protocol*, Distributed Computing **5** (1991), 83–94.

[PS96] B. Pierce and D. Sangiorgi, *Typing and subtyping for mobile processes*, Mathematical Structures in Computer Science **6** (1996), no. 5, 409–453.

[PS99] F. Pfenning and C. Schürmann, *System description: Twelf – a meta-logical framework for deductive systems*, Proc. CAD'99, LNAI, vol. 1632, Springer, 1999, pp. 202–206.

[Qua99] P. Quaglia, *The π-calculus: Notes on labelled semantics*, Bulletin of the EATCS (1999), no. 68, 104–114.

[QW98] P. Quaglia and D. Walker, *On encoding p-π in m-π*, Proc. FSTTCS'98, LNCS, vol. 1530, Springer, 1998, pp. 42–53.

[RE99] C. Röckl and J. Esparza, *Proof-checking protocols using bisimulations*, Proc. CONCUR'99, LNCS, vol. 1664, Springer, 1999, pp. 525–540.

[RE00] C. Röckl and J. Esparza, *On the mechanized verification of infinite systems*, Proc. SFB 342 Final Colloquium, Technische Universität München, 2000, pp. 31–52.

[Rey81] J. Reynolds, *The essence of ALGOL*, Algorithmic Languages, North-Holland, 1981, App. in vol. 2 of [OT97], pp. 345–372.

[RHB00] C. Röckl, D. Hirschkoff, and S. Berghofer, *Towards a formalization of π-calculus processes in higher order abstract syntax*, Tech. report, Technische Universität München, 2000.

[Röc99] C. Röckl, *First-order proofs for higher-order languages*, Proc. FBT'99, Utz, 1999, pp. 193–202.

[Röc00] C. Röckl, *Proving write invalidate cache coherence with bisimulations in Isabelle/HOL*, Proc. FBT'00, Shaker, 2000, pp. 69–78.

[RS99] C. Röckl and D. Sangiorgi, *A π-calculus process semantics of concurrent idealised ALGOL*, Proc. FOSSACS'99, LNCS, vol. 1578, Springer, 1999, pp. 306–321.

[Rut98] J. Rutten, *Automata and coinduction (an exercise in coalgebra)*, Proc. CONCUR'98, LNCS, vol. 1466, Springer, 1998, pp. 194–218.

[San92] D. Sangiorgi, *Expressing mobility in process algebras: First-order and higher-order paradigms*, Ph.D. thesis, University of Edinburgh, 1992.

[San95] D. Sangiorgi, *On the bisimulation proof method*, Proc. MFCS'95, LNCS, vol. 969, Springer, 1995, full version to appear in Mathematical Structures in Computer Science, pp. 479–488.

[San96a] D. Sangiorgi, *π-calculus, internal mobility, and agent-passing calculi*, Theoretical Computer Science (1996), no. 1&2, 235–274.

[San96b] D. Sangiorgi, *A theory of bisimulation of the π-calculus*, Acta Informatica **33** (1996), no. 1, 69–97.

[SM92] D. Sangiorgi and R. Milner, *The problem of weak bisimulation up-to*, Proc. CONCUR'92, LNCS, vol. 630, Springer, 1992, pp. 32–46.

[Sne95] J. Snepscheut, *The sliding-window protocol revisited*, Formal Aspects of Computing **7** (1995), 3–17.

[SOR93] N. Shankar, S. Owre, and J. Rushby, *The PVS proof checker: A reference manual*, Tech. report, SRI, 1993.

[Sta96] I. Stark, *A fully abstract domain model for the π-calculus*, Proc. LICS'96, IEEE Press, 1996, pp. 36–42.

[Ste97] C. Sterling, *Bisimulation, model checking and other games*, Notes for Mathfit Instructional Meeting on Games and Computation, Edinburgh, 1997.

[Tho90] B. Thomsen, *Calculi for higher order communicating systems*, Ph.D. thesis, Imperial College, 1990.

[Tur95] D. Turner, *The polymorphic pi-calculus: Theory and implementation*, Ph.D. thesis, University of Edinburgh, 1995.

[Wal95] D. Walker, *Objects in the π-calculus*, Information and Computation **116** (1995), 253–271.

[Wer94] B. Werner, *Une théorie des constructions inductives*, Ph.D. thesis, Université Paris, 1994.

[Yos96] N. Yoshida, *Graph types for monadic mobile processes*, Proc. FSTTCS'96, LNCS, vol. 1180, Springer, 1996, pp. 371–386.

Index

- Γ-closed
  - command, 70
  - context, 70
- α-conversion, 30
- β-reduction, 30
- π-calculus, 14
  - monadic, 15
  - polyadic, 15
- `claset()`, 29
- `simpset()`, 29

- actions, 6, 15
- adequacy, 57, 60
- Algol
  - concurrent (CIA), 63, 64
  - sequential (IA), 63, 64
- atomicity, 67

- binder, 29
- bisimulation
  - configuration, 70
  - up to expansion, 14, 19
  - up to expansion wrt. L, 103
- broadcast, 103, 105

- case-exhaustion, 25
- CCS
  - pure, 6
  - value-passing (VP), 8
- component, 91
- compositionality, 11, 13, 92, 98
- concurrent normal-form, 91
- configuration, 66, 72
  - locked, 69
- congruence, 11

- deep embedding, 32, 36

- elimination-rule, 7
- environment, 64
- equivalence
  - α-equivalence, 16, 41
  - observation, 11, 18, 71, 91
  - observation wrt. L, 92, 93
  - structural, 66
- exotic term, 32
- expansion, 13, 18
  - wrt. L, 102

- hypothetical judgement, 32

- induction, 31
  - rule, 25
  - structural, 31
- inductive set, 7, 22
- introduction-rule, 7, 22
- Isabelle
  - Isabelle/HOL, 21

- labelled transition-system, 7, 17
- labels, 6, 15
  - object, 8
  - subject, 8
- Leibnitz-equality, 50
- logical framework, 20
  - higher-order, 21

- meta-level, 21, 31
- meta-names, 36
- meta-variable, 31
- mobility, 14

- names, 6, 15, 36
  - bound, 15, 38
  - free, 15, 38, 47
  - fresh, 16, 38, 47
  - object, 14
  - subject, 14

- object-level, 21, 31
- object-names, 36
- object-variable, 31

operational correspondence, 74, 75

process-tree, 33

shallow embedding, 32, 36
snapback, 63
state-explosion, 11
substitution, 30, 72
syntax
  first-order, 29, 36
  higher-order (HOAS), 29, 32, 36

tactic, 27
  algebraic, 27
  classical, 27
tactical, 29
theorem-prover, 21
theory, 22
  `.ML`-file, 22
  `.thy`-file, 22
theory of contexts, 50
transition
  strong, 7, 17
  weak, 7, 17
type-system, 19
  I/O-types, 20
  linearity, 20, 79, 81
  sort, 19
  values, 20

well-formedness, 32, 48