Saarland University
Faculty of Natural Sciences and Technology I

Bachelor's Thesis

Compositional and Nameless Formalization of HOcore

Lukas Convent

Advisor:
Tobias Tebbi

Supervisor:
Prof. Dr. Gert Smolka

Reviewers:
Prof. Dr. Gert Smolka
Prof. Dr. Holger Hermanns

Submitted on August 17, 2016

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement in Lieu of an Oath:

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent:

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken, August 17, 2016

Abstract

The focus of this thesis lies on better formalization techniques for the higher-order process calculus HOcore.

HOcore allows different definitions of bisimilarity. They are usually given in a compositional manner, however, proofs about HOcore in the literature do not make use of their compositional structure. Looking at bisimilarity from a coinductive perspective, we follow the work of Damien Pous by using his notion of compatible functions to achieve compositional proofs for the soundness of up-to techniques and for closure properties of bisimilarity. As some closure properties of components depend on one another, we introduce the notion of conditional closedness as a criterion applicable to bisimilarities respecting these dependencies. Using de Bruijn indices, we avoid any side-conditions about disjointness of free variables. We introduce transitions which produce a context for each unguarded variable. By this means, we can analyze transitions of substituted processes in an elegant way, avoiding structural congruence and the separate treatment of guarded and unguarded variables.

We apply these techniques to HOcore and show soundness of the up-to-bisimilarity technique as well as substitutivity and congruence of IO bisimilarity. The resulting proof architecture, which is formalized in the proof assistant Coq, provides a better understanding of the different components and their dependencies.

Acknowledgements

First of all, I would like to thank my advisor Tobias Tebbi for the many hours of discussion and for pushing me on the right track again when things seemed hopeless. Working under his remarkable support taught me much about scientific research and was a very joyful experience. Then, I would like to thank Prof. Smolka for offering me this thesis and thereby allowing me to work in such a professional environment. I thank Prof. Smolka and Prof. Hermanns for introducing me to the topics of this thesis through their lectures as well as for reviewing it. I thank Agnes, Julian and Tobias for proofreading it and Yannick for helping me out with a nice LaTeX template. Finally, I would like to thank my family, my friends and especially Agnes for their support and many of these people for making studies in Saarbrücken such a lovely experience.

Contents

Chapter 1

Introduction

In this thesis, we focus on improved formalization techniques for higher-order process calculi, using the example of HOcore [6]. Forming the theoretical basis for the implementation of concurrent systems, process calculi allow the specification of communication between multiple processes operating in parallel. The characteristic of a higher-order process calculus like HOcore is that processes themselves are communicated between processes. This is a difference to first-order process calculi like the π-calculus [9] where channels (i.e. links between processes) are interchanged. Despite offering more modeling convenience, higher-order calculi do not add expressivity as they can be encoded in first-order calculi [13].

In order to handle higher-order substitutions (variables may be substituted by processes containing variables), we use *de Bruijn* indices to represent variables. In HOcore, processes are defined as follows

$$A, B \;::=\; \overline{a}\langle A\rangle \;\mid\; a.A \;\mid\; x \;\mid\; A \parallel B \;\mid\; \emptyset$$

Processes can communicate by sending ($\overline{a}$) and receiving (a) via channels. The communication behavior of a process is observable through labeled transitions, e.g. the transition $\overline{a}\langle A\rangle \xrightarrow{\overline{a}\langle A\rangle} \emptyset$ indicates that the process $\overline{a}\langle A\rangle$ can send a process A via channel a, thereby transitioning to $\emptyset$ (the empty process).

Two processes A and B are regarded as equivalent if they are bisimilar, this means intuitively that for each transition that A can make, B must be able to simulate this transition and vice versa. However, defining bisimilarity for a higher-order calculus is not straightforward: If we require that the capability of performing a transition like $\xrightarrow{\overline{a}\langle A\rangle}$ must be shared by bisimilar processes, we are overdiscriminating because a transition $\xrightarrow{\overline{a}\langle B\rangle}$ with bisimilar processes A and B can be seen as equivalent behavior, too.

In this thesis, we look at bisimilarity from a coinductive perspective: It is the greatest fixed-point of a monotone function over relations, which we call its functional. For different types of transitions we require different conditions, which leads to a functional consisting of multiple components. Each component of a bisimilarity is defined by a simulation functional, consider for example the functional describing higher-order input simulations.

$$s_{ho\_in}(\mathcal{R}) := \{(A, B) \mid \forall\, a, A'.\; A \xrightarrow{a} A' \quad\Rightarrow\quad \exists\, B'.\; B \xrightarrow{a} B' \;\wedge\; (A', B') \in \mathcal{R}\}$$

For a functional of bisimilarity, one can take the symmetric version of the intersection of the components' simulation functionals [11]. As we will focus on IO bisimilarity, consider the composed definition of its functional as an example.

$$\overleftrightarrow{s_{io}} := \overleftarrow{s_{ho\_out} \cap s_{ho\_in} \cap s_{var\_cxt}} = \overrightarrow{s_{ho\_out} \cap s_{ho\_in} \cap s_{var\_cxt}} \cap \overleftarrow{s_{ho\_out} \cap s_{ho\_in} \cap s_{var\_cxt}}$$

Despite the compositional structure of bisimilarity in HOcore, proofs about HOcore in the literature [6, 7] do not make use of it. This marks the starting point of our work.

Related work

HOcore was introduced by Lanese et al. when they studied the "expressiveness and decidability of higher-order process calculi" [6]. All results we have formalized for this thesis have been shown in their work through paper proofs.

There is already a formalization of "HOcore in Coq" by Maksimović et al. [7], which includes more results than we have given (coincidence of different bisimilarities, decidability, axiomatization). Their formalization techniques mainly differ in that they use the *locally named* approach for variable-binding and do not have a clear separation of proofs for different bisimilarity components. Another major difference is that in their formalization, substitutions are not performed in parallel, which requires lemmas about the irrelevance of the order of variable instantiation.

Parrow et al. have given a formalization of higher-order psi-calculi [10], which is based on Nominal Isabelle [18], a named-variable framework for Isabelle. They do not use higher-order substitutions, because variables are not substituted by processes, but by so-called handles which can then activate processes.

Our work is heavily relying on definitions and results of an order-theoretic study of up-to techniques as presented by Pous [11]. There, he shows how his notion of compatibility, which is closely related to Sangiorgi's notion of respectfulness [14], can be used to achieve compositional correctness[1] results for up-to techniques. As an accompaniment of a recent paper [12], Pous also gives a formalization of complete lattice theory in Coq[2]. Using Sangiorgi's notions of respectfulness and progressions [14], Hirschkoff gives a formalization of the (first-order) π-calculus in Coq based on de Bruijn indices [3].

Contribution

To our knowledge we seem to give the first formalization of a higher-order process calculus which makes use of the compositional structure of bisimilarity. Following the work of Pous [11], we use the notion of compatibility to achieve compositional proofs for the soundness of up-to techniques and for closure properties of bisimilarity. As some closure properties of components depend on one another, we introduce the notion of conditional closedness as a criterion applicable to components respecting these dependencies.

[1] We prove soundness [14] of up-to techniques, which implies correctness [11].

[2] We have not used the formalization for our development, because we have been only aware of it lately due to its recency.

Higher-order calculi require a special treatment of variable-binding, because the instantiation of free variables through a substitution must happen in a capture-avoiding way. In the named-variables setting of [6], side-conditions are used to require disjointness of variables. For the locally-named approach of [7], processes are accompanied by a well-formedness predicate ensuring the separation of local and global variables. We employ de Bruijn indices in combination with parallel substitutions, thereby avoiding both side conditions and well-formedness conditions. To see how this makes lemmas less clumsy, consider the example of substitutivity of transitions. The original lemma of [6] is stated as follows.

If $A \xrightarrow{\overline{a}\langle B\rangle} A'$ *with free variables in* $\tilde{C}$ *disjoint from the variables in* A, B *and* $\tilde{x}$, *then* $A\{\tilde{C}/\tilde{x}\} \xrightarrow{\overline{a}\langle B\{\tilde{C}/\tilde{x}\}\rangle} A'\{\tilde{C}/\tilde{x}\}$

Because instantiation of de Bruijn indices is capture-avoiding, we do not need to impose disjointness and can formulate the lemma as follows, using a substitution denoted by σ.

$$\textit{If } A \xrightarrow{\overline{a}\langle B\rangle} A', \textit{ then } A[\sigma] \xrightarrow{\overline{a}\langle B[\sigma]\rangle} A'[\sigma]$$

To analyze transitions of substituted processes, e.g. the output transition $A[\sigma] \xrightarrow{\overline{a}\langle B\rangle} A'$, we need to distinguish between a transition which A is already able to perform and one which is caused by a process substituted for an unguarded variable of A. The analysis is needed for the proof of substitutivity of bisimilarity, stating that if $A \sim B$, then also $A[\sigma] \sim B[\sigma]$. For the proof given in [6], there is a case analysis on the substituted variables, namely on whether they are guarded or unguarded, followed by reasoning via structural congruence. By introducing transitions which produce a context for each unguarded variable of a process, we obtain an elegant inversion lemma for a transition of the form $A[\sigma] \xrightarrow{\overline{a}\langle B\rangle} A'$, stating that A can either perform an input transition or a variable context transition. By this means, we avoid the separate treatment of guarded and unguarded variables through different lemmas. Additionally, as contexts preserve the structure of the original process, we avoid arguing about structural congruence.

Chapter 2

The Process Calculus

As our object of study, we choose the **HOcore** process calculus which was introduced by Lanese et al. [6]. **HO** stands for *higher order*, **core** denotes that it only includes a core set of constructs needed to establish higher-order communication.

For higher-order (process) calculi, a special role is accorded to variable-binding. This is because variables can be substituted for processes containing variables again, which obliges one to avoid their capturing. We use *de Bruijn* indices, to represent variables and to instantiate them in a capture-avoiding way [2].

2.1 Syntax

HOcore is made up of three disjoint sets of syntactic objects, namely processes (defined below), channels (distinct constants) and variables (represented by de Bruijn indices in $\mathbb{N}$). We use the letters A, B, C to range over processes, a, b, c to range over channels and x, y, z to range over variables. The set of **processes $\boldsymbol{Pr}$** is defined inductively by the following grammar.

A, B	$::= \ \overline{a}\langle A\rangle$	Output process
	$a.A$	Input-prefixed process
	x	Variable
	$A \parallel B$	Parallel composition
	$\emptyset$	Empty process

An output process $\overline{a}\langle A\rangle$ can send a process A through channel a. Its counterpart, an input-prefixed process of the form $a.A$, can receive a process from channel a and then continue with A. The input prefix of the form "$a.$" is the only variable-binding construct of the calculus and binds a free variable of A in a way which is described in the following paragraph.

We represent variables by de Bruijn indices. The position of a de Bruijn index, more specifically how deep it is nested within binders (i.e. within input prefixes), is relevant to determine the binder it refers to. For a de Bruijn index x, let n be the number of its enclosing binders. If $x < n$, then x refers to the $(x+1)$-th enclosing binder and represents a **bound variable**. If $x \geq n$, then x represents the **free variable** $(x-n)$. Note the difference between a free variable and an index representing

it; e.g. in the process $a.1$, the index 1 represents the free variable 0. A special type of free variable occurrences are unguarded variable occurrences. A variable occurrence is **unguarded** if it is not prefixed by an input prefix and not contained within an output process.

Finally, processes can operate in parallel when combined through parallel composition ($A \parallel B$) and there is the empty process $\emptyset$ which cannot proceed. Among other missing constructs like replication, choice and output prefixes, which are familiar e.g. from the π-calculus [9], note that there is no restriction operator. This has as a consequence that communication cannot be encapsulated (and therefore not be hidden from external observers).

2.2 Substitions

Substitutions are an integral part of a higher-order process calculus as they are used to instantiate variables by concrete processes and are therefore essential to implement communication. **Substitutions** (written σ, τ) are functions substituting free variables for processes, i.e. they are of type $\mathbb{N} \to Pr$. Because every variable is assigned to a process, all variables can be instantiated in parallel, as we describe further down. We call substitutions that replace variables for variables **renamings** (written ξ, ζ).

The great advantage of de Bruijn indices in comparison to named variables is that there is no need for α-conversion. Inlining a process into another one through substitution requires only shifting of free variables of the process to be inlined, which can be seen as a local adjustment.

In our Coq development, we use the Autosubst library [15] to deal with binders and substitutions. Autosubst is based on the σ-calculus [1], which is a rewriting system for parallel substitutions. For now, we define the formal constructs for substitutions as presented in [15] and refer to Chapter 9 for more details on how Autosubst facilitates reasoning about substitutions.

We define the **identity** and **shift** renamings:

$$id(x) := x$$
$$\uparrow(x) := x + 1$$

We can take the view that a substitution σ is an infinite list $\sigma(0), \sigma(1), ...$ This motivates the definition of a **cons** operation.

$$(A :: \sigma)(x) = \begin{cases} A & \text{if } x = 0 \\ \sigma(x-1) & \text{else} \end{cases}$$

The notation $\boldsymbol{A[\sigma]}$ stands for the **instantiation** operation: Every occurrence of a free variable x in A is replaced by $\sigma(x)$ in parallel (this is why we speak of *parallel* substitutions). We define instantiation together with **composition** of two substitutions

$$\frac{}{\overline{a}\langle A\rangle \xrightarrow{\overline{a}\langle A\rangle} \emptyset}\ \textsc{Out} \qquad \frac{}{a.A \xrightarrow{a} A}\ \textsc{In} \qquad \frac{}{x \xrightarrow{x} 0}\ \textsc{Var}$$

$$\frac{A \xrightarrow{\overline{a}\langle C\rangle} A' \quad B \xrightarrow{a} B'}{A \parallel B \xrightarrow{\tau} A' \parallel B'[C]}\ \textsc{SynL} \qquad \frac{A \xrightarrow{a} A' \quad B \xrightarrow{\overline{a}\langle C\rangle} B'}{A \parallel B \xrightarrow{\tau} A'[C] \parallel B'}\ \textsc{SynR}$$

$$\frac{A \xrightarrow{\overline{a}\langle C\rangle} A'}{A \parallel B \xrightarrow{\overline{a}\langle C\rangle} A' \parallel B}\ \textsc{OutParL} \qquad \frac{B \xrightarrow{\overline{a}\langle C\rangle} B'}{A \parallel B \xrightarrow{\overline{a}\langle C\rangle} A \parallel B'}\ \textsc{OutParR}$$

$$\frac{A \xrightarrow{a} A'}{A \parallel B \xrightarrow{a} A' \parallel B[\uparrow]}\ \textsc{InParL} \qquad \frac{B \xrightarrow{a} B'}{A \parallel B \xrightarrow{a} A[\uparrow] \parallel B'}\ \textsc{InParR}$$

$$\frac{A \xrightarrow{\tau} A'}{A \parallel B \xrightarrow{\tau} A' \parallel B}\ \textsc{SynParL} \qquad \frac{B \xrightarrow{\tau} B'}{A \parallel B \xrightarrow{\tau} A \parallel B'}\ \textsc{SynParR}$$

$$\frac{A \xrightarrow{x} A'}{A \parallel B \xrightarrow{x} A' \parallel B[\uparrow]}\ \textsc{VarParL} \qquad \frac{B \xrightarrow{x} B'}{A \parallel B \xrightarrow{x} A[\uparrow] \parallel B'}\ \textsc{VarParR}$$

Figure 2.1: Transitions

and the **up-operator** $\Uparrow$ in a mutually recursive manner.

$$\begin{aligned}(\overline{a}\langle A\rangle)[\sigma] &= \overline{a}\langle A[\sigma]\rangle \\ (a.A)[\sigma] &= a.(A[\Uparrow \sigma]) \\ x[\sigma] &= \sigma(x) \\ (A \parallel B)[\sigma] &= A[\sigma] \parallel B[\sigma] \\ \emptyset[\sigma] &= \emptyset\end{aligned} \qquad \begin{aligned}(\sigma \circ \tau)(x) &= \sigma(x)[\tau] \\ \Uparrow \sigma &= 0 :: (\sigma \circ \uparrow)\end{aligned}$$

Note that by using $\Uparrow$, instantiation under a binder preserves the 0 indices (since they are bound) and shifts all other indices up.

Instead of writing $A[B :: id]$, we use the shorthand notation $\boldsymbol{A[B]}$.

2.3 Transitions

Transitions represent the observable behavior of processes by relating a source process via an observable action to a target process. Our definition of HOcore has four types of transitions, namely **output** transitions, **input** transitions, $\boldsymbol{\tau}$**-transitions** and **variable context** transitions. They are defined inductively by the inference rules in Figure 2.1. By including variable context transitions, our definition of HOcore differs from the ones in [6, 7]. A further elaboration on the differences of definitions is given in Chapter 8.

An output process of the form $\overline{a}\langle A\rangle$ can send a process A on channel a and is stuck afterwards (Out). An input-prefixed process of the form $a.A$ can signal that it is ready to receive a process via channel a (In). Note that an input transition strips

off a binder: Occurrences of the free 0 variable in A are ready to be replaced by an incoming process.

If there is an outputting process A and an inputting process B operating in parallel on the same channel, they can synchronize (SynL and SynR). Synchronization as defined by SynL (and analogously by SynR) means that both processes evolve according to their transitions, with the emitted process C being handed over to the target process B' of B: This results in the process $B'[C]$, having all occurrences of the free 0 variable in B' replaced by C and all other free variables lowered, so that they are at the same level as the free variables in A'.

A process can make a variable context transition for each unguarded variable occurrence, thereby producing a context. We define a **context** (written C) as a process which contains exactly one free 0 variable, representing the context's hole. A context C can be filled by instantiating it with a special kind of substitution: Filling C with a process A results in $C[A]$. The combination of encoding contexts as processes and producing them via variable context transitions provides for an elegant handling of substituted processes. We go into more detail about this in Section 6.2.

Transitions can happen below the parallel operator according to the ParL and ParR rules. Noticeable about the InParL and VarParL rules (and analogously for their ParR-counterparts) is that for the composed target process, the free variables of the passive process B are shifted up. This is needed because the target process A' has introduced new 0 variables (exactly one in case of VarParL) and shifted up all other variables, requiring an alignment of the variables of the passive process B.

The set of processes Pr, transitions and the observable actions together form a *labeled transition system* (LTS), which corresponds to an infinite directed graph connecting processes via their transitions. We point out this structure because bisimulations are commonly defined on LTS.

2.3.1 Substitutivity and Preservation of Renaming

If a process A can make an output, an input or a τ-transition, then so can $A[\sigma]$. For variable context transitions, this holds only for renamings ξ.

Lemma 2.1

$$\frac{A \xrightarrow{\overline{a}\langle B\rangle} A'}{A[\sigma] \xrightarrow{\overline{a}\langle B[\sigma]\rangle} A'[\sigma]} \qquad \frac{A \xrightarrow{a} A'}{A[\sigma] \xrightarrow{a} A'[\Uparrow \sigma]} \qquad \frac{A \xrightarrow{\tau} A'}{A[\sigma] \xrightarrow{\tau} A'[\sigma]} \qquad \frac{A \xrightarrow{x} A'}{A[\xi] \xrightarrow{\xi(x)} A'[\Uparrow \sigma]}$$

Proof By induction over the transition. ∎

If a renamed process $A[\xi]$ can make an output, an input, a τ- or a variable context transition, then so can A.

Lemma 2.2

$$\frac{A[\xi] \xrightarrow{\overline{a}\langle B\rangle} C}{\begin{array}{l}\exists\, B', C'.\ A \xrightarrow{\overline{a}\langle B'\rangle} C' \\ B = B'[\xi] \quad C = C'[\xi]\end{array}} \qquad \frac{A[\xi] \xrightarrow{a} C}{\begin{array}{l}\exists\, C'.\ A \xrightarrow{a} C' \\ C = C'[\Uparrow \xi]\end{array}} \qquad \frac{A[\xi] \xrightarrow{\tau} C}{\begin{array}{l}\exists\, C'.\ A \xrightarrow{\tau} C' \\ C = C'[\xi]\end{array}} \qquad \frac{A[\xi] \xrightarrow{x} C}{\begin{array}{l}\exists\, C', y.\ A \xrightarrow{y} C' \\ C = C'[\Uparrow \xi] \quad x = \xi(y)\end{array}}$$

Proof By induction over the transition. ∎

Chapter 3

Bisimulations

As in any process calculus, we need to specify what it means for two processes to be equivalent. Based upon the notion of bisimulation, the appropriate equivalence describing behavioral equivalence is bisimilarity [8]. Bisimilarity is a coinductive notion and can be defined in terms of a monotone function. As hinted at in the introduction, there are different suitable functions. We take a closer look on IO bisimilarity, for which we will prove some properties in the following chapters under consideration of its compositional character.

We present the concept of bisimulation and bisimilarity by using the example of higher-order input bisimulations, which we already referred to in the introduction. On our LTS of processes, transitions and observable actions (as identified in Section 2.3), we define the first bisimulation.

Definition 3.1 *A relation $\mathcal{R} \in Pr \times Pr$ is a **higher-order input bisimulation** if for all $(A, B) \in \mathcal{R}$ holds:*

- *For any transition $A \xrightarrow{a} A'$, there exists B' s.t. $B \xrightarrow{a} B'$ and $(A', B') \in \mathcal{R}$*
- *For any transition $B \xrightarrow{a} B'$, there exists A' s.t. $A \xrightarrow{a} A'$ and $(A', B') \in \mathcal{R}$*

Two processes A, B are **higher-order-input-bisimilar** ($A \sim_{ho\_in} B$) if there exists a bisimulation $\mathcal{R}$ s.t. $(A, B) \in \mathcal{R}$.

3.1 Coinductive Definition

Bisimilarity is a coinductive notion, so it can also be defined in terms of fixed-point theory [8]. To do so, we identify the complete lattice $\langle Pr \times Pr, \subseteq, \bigcup \rangle$ on the set of all process relations $Pr \times Pr$ with set union $\bigcup$ as its join operation. The meet operation is defined in terms of the join operation and is identical to set intersection $\bigcap$. We use the letters $\mathcal{R}, \mathcal{S}$ to range over relations. We write $\mathcal{R} \cdot \mathcal{S}$ for the **relation composition** of $\mathcal{R}$ and $\mathcal{S}$, 1 for the **identity relation**, $\bot$ for the **empty relation** and $\top$ for the **universal relation**. The notation id stands for the **identity function** $\lambda\, \mathcal{R}.\, \mathcal{R}$ and the notation $\hat{\mathcal{S}}$ for the **constant function** $\lambda\, \mathcal{R}.\, \mathcal{S}$. We extend set inclusion $\subseteq$ in a pointwise manner to functions.

For a function b over relations, we call a relation $\mathcal{R}$ with $\mathcal{R} \subseteq b(\mathcal{R})$ a **post-fixed-point** of b. For a monotone function b, we denote the union of all post-fixed-points

$\bigcup\{\mathcal{R} \mid \mathcal{R} \subseteq b(\mathcal{R})\}$ by $\boldsymbol{\nu b}$. A bisimilarity $\sim$ can be characterized by a monotone function b s.t.

$$\begin{array}{rcl} \mathcal{R} \text{ is a bisimulation} & \Leftrightarrow & \mathcal{R} \subseteq b(\mathcal{R}) \ (\mathcal{R} \text{ is a post-fixed-point of } b). \\ \sim & = & \nu b \end{array}$$

We call a monotone function b the **functional** of νb. The functional corresponding to higher-order input bisimilarity (Definition 3.1) is as follows.

$$b_{ho\_in}(\mathcal{R}) := \{(A,B) \mid {\scriptstyle \forall\, a, A'.\ A \xrightarrow{a} A' \Rightarrow \exists B'.B \xrightarrow{a} B' \wedge (A',B') \in \mathcal{R}} \ \wedge\ {\scriptstyle \forall\, a, B'.\ B \xrightarrow{a} B' \Rightarrow \exists A'.A \xrightarrow{a} A' \wedge (A',B') \in \mathcal{R}}\}$$

The connection to greatest fixed-points is given by a theorem of Knaster and Tarski [17].

Theorem 3.2 (Knaster-Tarski) *For a monotone function b, νb is the greatest fixed-point of b.*

By formulating bisimilarity through greatest fixed-points of functionals, we can combine functionals to obtain new ones, i.e. they can be composed, transposed, pointwisely intersected and so on. The great advantage of this coinductive perspective is that next to combining the functionals themselves, there are techniques to combine properties of them at the same time. Before we go into detail on this from Chapter 4 on, we describe in the following how we can combine functionals.

3.2 From Simulations to Bisimulations

As already displayed in the term "*bi*simulation", the capability of simulating the other process (i.e. every transition of A can be matched by a transition of B) needs to be preserved on *both* sides. Simulations can be characterized by a simulation functional in the same way as bisimulations can be characterized by a bisimulation functional. We follow Pous [11] in working with a simulation functional which can then be lifted to a bisimulation functional, using intersection and transposition.

We define **transposition** of relations as $\overline{\mathcal{R}} := \{(A,B) \mid (B,A) \in \mathcal{R}\}$ and extend this definition to functions as follows: $\overline{f}(\mathcal{R}) := \overline{f(\overline{\mathcal{R}})}$. We extend intersection $\cap$ of relations in a pointwise manner to functions. Finally, we can introduce the notion of **symmetrization** which captures the lifting from a simulation functional to a bisimulation functional: $\overleftrightarrow{s} := s \cap \overline{s}$. Considering the example of higher-order input bisimulations, we obtain the following equivalent formulation of its functional.

$$b_{ho\_in}(\mathcal{R}) = \overleftrightarrow{s_{ho\_in}} \qquad \text{with} \qquad s_{ho\_in}(\mathcal{R}) = \{(A,B) \mid {\scriptstyle \forall A' A \xrightarrow{a} A' \Rightarrow \exists B'.B \xrightarrow{a} B' \wedge (A',B') \in \mathcal{R}}\}$$

There is another way to obtain bisimulations in terms of a simulation functional. In [6] and [7], only symmetric bisimulations are considered. This is convenient, because symmetric bisimulations coincide with symmetric simulations. For our formalization, we have not observed any particular advantage of using general bisimulations (through the symmetrization operator) over symmetric bisimulations (through symmetric simulations). However, when it comes to finding a bisimulation in order to show that two concrete processes are bisimilar, requiring a symmetric bisimulation limits the number of candidates. This seems unnatural to us, letting us choose to work with the symmetrization operator in our formalization.

3.3 Bisimulations in HOcore

For the definition of bisimilarity in a first-order process calculus (like the π-calculus), we require that a transition gets mirrored by the exact same transition. For higher-order process calculi, this strict requirement is insufficient: Because $A \parallel B \sim B \parallel A$, we also expect $\overline{a}\langle A \parallel B\rangle \sim \overline{a}\langle B \parallel A\rangle$. The solution to this is to require emitted processes not to be syntactically equal, but bisimilar.

Multiple definitions of bisimilarity for HOcore, among them IO bisimilarity, turn out to coincide [6]. All definitions have in common that they can be formulated in terms of intersections of different simulation functionals.

In this thesis, we will focus on IO bisimilarity, a bisimilarity for which Lanese et al. [6] have given a direct decision procedure.

3.4 IO Bisimilarity

IO bisimilarity is a combination of three components, which are defined by the **higher-order output**, **higher-order input** and **variable context** functionals.

1. $(A, B) \in \boldsymbol{s_{ho\_out}}(\mathcal{R})$ iff each transition $A \xrightarrow{\overline{a}\langle A''\rangle} A'$ implies that there are B', B'' s.t. $B \xrightarrow{\overline{a}\langle B''\rangle} B'$ with $(A', B') \in \mathcal{R}$ and $(A'', B'') \in \mathcal{R}$
2. $(A, B) \in \boldsymbol{s_{ho\_in}}(\mathcal{R})$ iff each transition $A \xrightarrow{a} A'$ implies that there is a B' s.t. $B \xrightarrow{a} B'$ with $(A', B') \in \mathcal{R}$
3. $(A, B) \in \boldsymbol{s_{var\_cxt}}(\mathcal{R})$ iff each transition $A \xrightarrow{x} A'$ implies that there is a B' s.t. $B \xrightarrow{x} B'$ with $(A', B') \in \mathcal{R}$

We define $\boldsymbol{s_{io}} := s_{ho\_out} \cap s_{ho\_in} \cap s_{var\_cxt}$ and $\boldsymbol{\sim_{io}} := \nu \overleftrightarrow{s_{io}}$.

The first two components, s_{ho_out} and s_{ho_in}, are defined as in [6]. For every output and input transition of a simulated process, there must be a corresponding transition such that the emitted processes and target processes are in the simulation, too.

The third component, s_{var_cxt}, requires that every produced context can be simulated. It thereby also requires that every unguarded variable of A is contained in B, too.

Proposition 3.3 *For a process A and a variable x, there exists an unguarded occurrence of x in A iff there exists a transition $A \xrightarrow{x} A'$ with a context A'.*

By including the bisimulation of variable context transitions (through s_{var_cxt}), our definition of IO bisimilarity differs from the ones of [6, 7][1]. However, in Section 8.2 we show that our definition coincides with the one of [7]. The bisimulation functional $\overleftrightarrow{s_{var\_cxt}}$ requires two bisimilar processes to have the same number of unguarded occurrences for each variable. Requiring this is necessary to obtain congruence of IO bisimilarity: E.g., we want $0 \nsim 1$ and $0 \nsim 0 \parallel 0$, because they behave differently in certain contexts.

[1] The differences are elaborated on in Chapter 8.

Chapter 4

Up-to Techniques

Two processes are bisimilar if there exists a bisimulation which relates them. Because such bisimulations can be very large and hard to describe, up-to techniques have been introduced (e.g. in [8, 14]). For a functional b describing a bisimilarity $\sim_b$, an up-to technique enhances b in such a way that the greatest fixed-point (bisimilarity) is maintained but more post-fixed-points (bisimulations) exist. This enhancement consists of composing the functional with an **up-to function**.

Definition 4.1 *For a functional s, a function f is **s-sound** if*

$$\forall \mathcal{R}. \quad \mathcal{R} \subseteq s(f(\mathcal{R})) \quad \Rightarrow \quad \mathcal{R} \subseteq \nu s$$

Suppose we want to show $A \sim_b B$ for a bisimulation functional b. Now, a b-sound function f allows a new proof technique for showing bisimilarity: Instead of finding a b-bisimulation $\mathcal{R}$ with $(A, B) \in \mathcal{R}$, it suffices to find a $(b \circ f)$-bisimulation $\mathcal{R}$ with $(A, B) \in \mathcal{R}$. Such a relation is then called a **bisimulation-up-to-f**. Usually f is extensive ($id \subseteq f$), which allows for smaller relations $\mathcal{R}$. Our proof obligation has thus been weakened from $\mathcal{R} \subseteq b(\mathcal{R})$ to $\mathcal{R} \subseteq b(f(\mathcal{R}))$.

Unfortunately, soundness does not have good closure properties. E.g., just because f is an s_1-sound and s_2-sound up-to function, this does not necessarily mean it is also an $(s_1 \cap s_2)$-sound up-to function, too. As a remedy, we follow the ideas of Pous [11] and use his notion of compatibility as a soundness criterion with good closure properties. We give a method for showing soundness of the up-to-bisimilarity technique for compositional definitions of bisimilarity. The purpose is to demonstrate how compatibility allows compositional proofs of up-to techniques for compositional definitions of bisimilarity. For this, the up-to-bisimilarity technique serves as an example.

For the rest of the chapter, we fix a (monotone) functional s.

4.1 Compatibility

Because s-soundness has bad closure properties, compatibility is introduced by Pous in [11] as a sufficient condition for s-soundness with good closure properties[1]. We

[1] To be more precise, in [11] compatibility is introduced as a sufficient condition for s-correctness, which implies s-soundness.

$$\frac{f \text{ is } s\text{-compatible} \quad g \text{ is } s\text{-compatible}}{(f \circ g) \text{ is } s\text{-compatible}}$$ (i) Closure under composition

$$\frac{f_1 \text{ is } s\text{-compatible} \quad f_2 \text{ is } s\text{-compatible}}{(f_1 \cup f_2) \text{ is } s\text{-compatible}}$$ (ii) Closure under union

$$\frac{f \text{ is } s_1\text{-compatible} \quad f \text{ is } s_2\text{-compatible}}{f \text{ is } (s_1 \cap s_2)\text{-compatible}}$$ (iii) Closure under intersection

$$\frac{f \text{ is } s\text{-compatible}}{\overline{f} \text{ is } \overline{s}\text{-compatible}}$$ (iv) Closure under transposition

$$\frac{f \text{ symmetric} \quad f \text{ is } s\text{-compatible}}{f \text{ is } \overleftrightarrow{s}\text{-compatible}}$$ (v) Closure under symmetrization

Figure 4.1: Closure properties of compatibility

define compatibility in a slightly more general way by extending its applicability from monotone functions to any functions (not necessarily monotone)[2].

Definition 4.2 *A function f is **s-compatible** if*

$$\forall \mathcal{R}, \mathcal{S}. \quad \mathcal{R} \subseteq s(\mathcal{S}) \quad \Rightarrow \quad f(\mathcal{R}) \subseteq s(f(\mathcal{S}))$$

It turns out that many interesting up-to techniques can be proven compatible [11]. The strength of compatibility are its closure properties, whose proofs are straightforward. They allow the combination and transposition of different up-to functions and functionals (Figure 4.1 (i) - (iv)). As a corollary, we can lift symmetric up-to techniques from a simulation functional s to its corresponding bisimulation functional $\overleftrightarrow{s}$ (Figure 4.1 (v)). A function f is **symmetric** if $f = \overline{f}$.

4.1.1 Comparison to other Soundness Criteria

Before we show that compatibility implies soundness, we compare the present definition of compatibility to the notion of respectfulness [14] and to the original definition of compatibility given by Pous [11].

Compatibility is based on the notion of respectfulness. The notion of respectfulness was introduced by Sangiorgi [14] and is very similar to the notion of compatibility, both serving as a soundness criterion and allowing for compositional results. A function f is **s-respectful** if

$$\forall \mathcal{R}, \mathcal{S}. \quad \mathcal{R} \subseteq \mathcal{S} \ \wedge \ \mathcal{R} \subseteq s(\mathcal{S}) \quad \Rightarrow \quad f(\mathcal{R}) \subseteq s(f(\mathcal{S}))$$

The definition of respectfulness only differs from the definition of compatibility in the additional assumption $\mathcal{R} \subseteq \mathcal{S}$. An observation of Pous [12] can be carried over to the

[2] The idea of extending compatibility to (not necessarily monotone) functions was proposed by Steven Schäfer in private communication.

present definition of compatibility: A function f is s-respectful iff f is $\dot{s}$-compatible, with $\dot{s} := s \cap id$ being the **decreasing version of a function** s.

By the Pous' definition of compatibility [11], a monotone function f is said to be s-compatible if $f \circ s \subseteq s \circ f$. Besides having a different appearance, our definition of compatibility mainly differs from Pous' definition in that it is stated for (not necessarily monotone) functions whereas Pous' definition is stated for monotone functions. As Pous [12] observes, the two versions of compatibility coincide for monotone f.

Proposition 4.3 *For a monotone function f, f is s-compatible iff $f \circ s \subseteq s \circ f$.*

Although most interesting up-to functions are monotone, the present definition gives us the advantage of not having to show monotonicity.

4.1.2 Compatibility implies Soundness

We now show that compatibility implies soundness. As a proof method, we use the companion introduced by Pous [12]. The **companion c_s** of a functional s is defined as the pointwise union of all compatible monotone functions and is a monotone function itself.

$$c_s := \bigcup \{f \mid f \text{ is } s\text{-compatible} \ \wedge \ f \text{ monotone}\}$$

As shown by Pous in [12], the companion c_s preserves the greatest fixed-point νs.

Lemma 4.4 $\nu(s \circ c_s) = \nu s$.

We can follow:

Proposition 4.5 *c_s is s-sound.*

For a functional s, the companion c_s contains all compatible *monotone* functions by definition. To show that it also contains all compatible *(not necessarily monotone)* functions, we can use the method of monotonization. We define the (upper) **monotonization** of a function f as

$$\ulcorner f \urcorner(\mathcal{R}) := \bigcup_{\mathcal{S} \subseteq R} f(\mathcal{S})$$

The central idea is the following:

Lemma 4.6 *If a function f is s-compatible, then $\ulcorner f \urcorner$ is also s-compatible.*

Proof Let $\mathcal{R}$, $\mathcal{S}$ be relations with $\mathcal{R} \subseteq s(\mathcal{S})$. We have to show $\ulcorner f \urcorner(\mathcal{R}) \subseteq s(\ulcorner f \urcorner(\mathcal{S}))$, which is equivalent to showing

$$\bigcup_{\mathcal{T} \subseteq \mathcal{R}} f(\mathcal{T}) \quad \subseteq \quad s(\ulcorner f \urcorner(\mathcal{S}))$$

Let $\mathcal{T}$ be a relation with $\mathcal{T} \subseteq \mathcal{R}$. It suffices to show $f(\mathcal{T}) \subseteq s(\ulcorner f \urcorner(\mathcal{S}))$:

$$\begin{aligned} f(\mathcal{T}) &\subseteq s(f(\mathcal{S})) && f \text{ is } s\text{-compatible} \\ &\subseteq s(\ulcorner f \urcorner(\mathcal{S})) && \ulcorner \cdot \urcorner \text{ is extensive } (f \subseteq \ulcorner f \urcorner) \end{aligned}$$

■

Because monotonization is an extensive operation ($f \subseteq \ulcorner f \urcorner$), every s-compatible (not necessarily monotone) function is contained in c_s. We use this fact for the main result of this section.

Theorem 4.7 *For a functional s, every s-compatible function is s-sound.*

Proof Assume a functional s, an s-compatible function f and a relation $\mathcal{R}$ with $\mathcal{R} \subseteq s(f(\mathcal{R}))$. We have to show $\mathcal{R} \subseteq \nu s$:

$\mathcal{R} \subseteq s(f(\mathcal{R}))$	By assumption
$\subseteq s(\ulcorner f \urcorner(\mathcal{R}))$	$\ulcorner \cdot \urcorner$ is extensive ($f \subseteq \ulcorner f \urcorner$)
$\subseteq s(c_s(\mathcal{R}))$	$\ulcorner f \urcorner$ is monotone and s-compatible (by Lemma 4.6)
$\subseteq \nu(s \circ c_s)$	$\mathcal{R}$ is a post-fixed-point of $s \circ c_s$
$= \nu s$	By Lemma 4.4

■

4.2 Up-to-Bisimilarity

The **up-to-bisimilarity** technique is the function $f_\sim(\mathcal{R}) := \sim \cdot \mathcal{R} \cdot \sim$. It extends a relation by adding all pairs of processes which are bisimilar to the ones originally related. In [6, 7], the authors use up-to techniques, among them up-to-bisimilarity [7] and up-to structural congruence (an "important tool" [7]). We have not used up-to techniques in our formalization. In general their significance for dealing with a process calculus however is indisputable, making their formalization worthwhile.

We establish the soundness of the up-to-bisimilarity technique for a compositional definition of bisimilarity by showing its compatibility. To this end, we define **pointwise relation composition** as $(f \mathbin{\hat{\cdot}} g)(\mathcal{R}) := f(\mathcal{R}) \cdot g(\mathcal{R})$ and use it to give a reformulation of $f_\sim$.

$$\begin{aligned} & f_\sim(\mathcal{R}) &&= \sim \cdot \mathcal{R} \cdot \sim \\ \Leftrightarrow \quad & f_\sim &&= \hat{\sim} \mathbin{\hat{\cdot}} id \mathbin{\hat{\cdot}} \hat{\sim} \end{aligned}$$

Then, for a functional s, we define the property $\forall \mathcal{R}, \mathcal{S}.\ s(\mathcal{R}) \cdot s(\mathcal{S}) \subseteq s(\mathcal{R} \cdot \mathcal{S})$ as **preservation of transitivity**[3]. For the compatibility proof of $f_\sim$, we use two results of Pous [11] which can be directly applied to the reformulation of $f_\sim$. First, s-compatibility is closed under pointwise relation composition if s preserves transitivity.

Proposition 4.8 (Compatibility is closed under pointwise rel. composition)

$$\frac{\forall \mathcal{R}, \mathcal{S}.\ s(\mathcal{R}) \cdot s(\mathcal{S}) \subseteq s(\mathcal{R} \cdot \mathcal{S}) \quad f_1 \textit{ is } s\textit{-compatible} \quad f_2 \textit{ is } s\textit{-compatible}}{(f_1 \mathbin{\hat{\cdot}} f_2) \textit{ is } s\textit{-compatible}}$$

Second, two types of functions, which both occur in $f_\sim$, are always compatible.

[3] Pous [11] introduces this property as a part of "preservation of the monoid $\langle Pr \times Pr, \cdot, 1 \rangle$"

Proposition 4.9 *The identity function id is s-compatible. If $\mathcal{R} \subseteq s(\mathcal{R})$, then the constant function $\hat{\mathcal{R}}$ is s-compatible.*

Preservation of transitivity has good closure properties so that we can apply Proposition 4.8 to a composed functional s.

Proposition 4.10

$$\frac{\begin{array}{ll} & s_1 \text{ preserves transitivity} \\ s = s_1 \cap s_2 & s_2 \text{ preserves transitivity} \end{array}}{s \text{ preserves transitivity}} \qquad \frac{s \text{ preserves transitivity}}{\overline{s} \text{ preserves transitivity}}$$

Finally, we can use Proposition 4.8, Proposition 4.9 and Proposition 4.10 to obtain our main result.

Theorem 4.11 (Compatibility of up-to-bisimilarity) *Given a functional s of the form $s = s_1 \cap ... \cap s_n$, let $\sim$ be defined as $\nu^{\overleftrightarrow{s}}$. If each s_i preserves transitivity, then $f_\sim$ is $\overleftrightarrow{s}$-compatible.*

Proof First, we note that from the premises $s_i(\mathcal{R}) \cdot s_i(\mathcal{S}) \subseteq s_i(\mathcal{R} \cdot \mathcal{S})$ we obtain $\overleftrightarrow{s}(\mathcal{R}) \cdot \overleftrightarrow{s}(\mathcal{S}) \subseteq \overleftrightarrow{s}(\mathcal{R} \cdot \mathcal{S})$: This is due to the closure properties given by Proposition 4.10.

Second, by $\overleftrightarrow{s}(\mathcal{R}) \cdot \overleftrightarrow{s}(\mathcal{S}) \subseteq \overleftrightarrow{s}(\mathcal{R} \cdot \mathcal{S})$, we will show that $f_\sim$ is $\overleftrightarrow{s}$-compatible: By Proposition 4.8, showing compatibility of the two components $\hat{\sim}$ and id proves compatibility of the composition $\hat{\sim} \mathbin{\hat{:}} id \mathbin{\hat{:}} \hat{\sim}$. This follows from Proposition 4.10. ■

Because the components of IO bisimilarity preserve transitivity, we obtain by Theorem 4.11 correctness of the up-to-bisimilarity technique for it.

Proposition 4.12 *s_{ho_out}, s_{ho_in} and s_{var_cxt} preserve transitivity.*

Chapter 5

Closure Properties of Bisimilarity

Many properties of bisimilarity are closure properties. A bisimilarity $\sim$ is **closed** under a function f if $f(\sim) \subseteq \sim$. For example, substitutivity can be formulated as a closure property.

Definition 5.1 (Substitutivity) *A bisimilarity $\sim$ is substitutive if it is closed under f_{subst} with $f_{subst}(\mathcal{R}) := \{(A[\sigma], B[\sigma]) \mid (A, B) \in \mathcal{R},\ \sigma \text{ substitution}\}$*

Similar to soundness of up-to techniques, closure properties of bisimilarity are not compositional either. E.g., just because νs_1 and νs_2 are both closed under f, this does not necessarily mean that $\nu(s_1 \cap s_2)$ is closed under f, too.

In the following, we present two ways of proving a closure property for a compositional definition of bisimilarity. First, we show how compatibility can help us. Because there are cases where compatibility is too strong, we introduce the notion of conditional closedness as an alternative criterion, which still has good closure properties.

5.1 Compatibility implies Closedness

Let νs be a bisimilarity defined through a functional s. When trying to prove that it is closed under a function f, i.e. $f(\nu s) \subseteq \nu s$, we can use compatibility again and therefore profit from its closure properties.

Lemma 5.2 (Compatibility implies closedness)

$$\frac{f \text{ is } s\text{-compatible}}{f(\nu s) \subseteq \nu s}$$

Proof It suffices to show $f(\nu s) \subseteq s(f(\nu s))$. We have this by definition of compatibility with $\mathcal{R} = \mathcal{S} = \nu s$. ∎

Let a functional s have the form $s = s_1 \cap ... \cap s_n$ and let f be symmetric. We can prove that $\nu(\overleftrightarrow{s})$ is closed under f: Having shown f to be s_i-compatible for every i, by the closure properties of compatibility we have $\overleftrightarrow{s}$-compatibility of f (by Figure 4.1 (iii), (v)). Finally, the claim follows from Lemma 5.2.

5.2 Conditional Closure Properties

For some properties, components require certain conditions to hold for a bisimilarity which they are part of. Consider this example: In order to show that a bisimilarity $\sim$ containing the component s_{ho_out} is substitutive, we have to rely on $\sim$ being both reflexive and only relating processes which have the same unguarded variables. The conditions for $\sim$ can be divided into two categories, **minimum conditions** and **maximum conditions**. Minimum conditions correspond to pre-fixed-point properties and maximum conditions to post-fixed-point properties.

Reflexivity is an example of a minimum condition: $\sim$ must contain *at least* the identity relation 1. The requirement of having the same unguarded variables is an example of a maximum condition: $\sim$ may contain *at most* pairs possessing the same unguarded variables.

Proving the substitution closure to be s_{ho_out}-compatible is not possible, because compatibility does not let us impose any minimum/maximum condition as it is needed in this case. Thus we need to find a different criterion which incorporates the minimum and maximum conditions but remains compositional at the same time.

5.2.1 Conditional Closedness

To show closedness of bisimilarities fulfilling certain conditions, we introduce the notion of **conditional closedness**.

Definition 5.3 *For a functional s and functions f, g_1, g_2, we call s* ***f-closed above g_1 and below g_2*** *if:*

$$\forall \mathcal{R}. \quad g_1(\mathcal{R}) \subseteq \mathcal{R} \;\wedge\; \mathcal{R} \subseteq g_2(\mathcal{R}) \;\wedge\; \mathcal{R} \subseteq s(\mathcal{R}) \quad \Rightarrow \quad f(\mathcal{R}) \subseteq s(f(\mathcal{R}))$$

For "f-closed above g_1 and below g_2" we write "$\boldsymbol{f}$**-closed**$^{\boldsymbol{g_2}}_{\boldsymbol{g_1}}$". We also call g_1 (g_2) the **minimum** (**maximum**) **condition** of the statement. Conditional closedness differs in two ways from compatibility: First, the two additional side conditions confine the statement's applicability to relations fulfilling a minimum and a maximum condition, expressed by g_1 and g_2. Second, the distinction between $\mathcal{R}$ and $\mathcal{S}$ is canceled as it is not needed to show the closure property. Once we have established conditional f-closedness of s, we can separately show $g_1(\nu s) \subseteq \nu s$ and $\nu s \subseteq g_2(\nu s)$ and obtain our desired result, closedness of νs under f.

Lemma 5.4 (Conditional closedness implies closedness)

$$\frac{s \text{ is } f\text{-closed}^{g_2}_{g_1} \quad g_1(\nu s) \subseteq \nu s \quad \nu s \subseteq g_2(\nu s)}{f(\nu s) \subseteq \nu s}$$

Proof It suffices to show $f(\nu s) \subseteq s(f(\nu s))$. We have this by definition of conditional closedness with $\mathcal{R} = \nu s$. ∎

As presented in Figure 5.1, conditional closedness has closure properties which are very similar to those of compatibility (Figure 4.1) and whose proofs are straightforward.

Proving that a bisimilarity $\nu^{\leftrightarrow}_{s}$ with $s := s_1 \cap ... \cap s_n$ is closed under a symmetric function f can thus be done in a compositional way: By showing every s_i to be

$$\frac{g_1 \subseteq g_1' \quad g_2 \supseteq g_2' \quad s \text{ is } f\text{-closed}_{g_1}^{g_2}}{s \text{ is } f\text{-closed}_{g_1'}^{g_2'}}$$ (i) Consolidating conditions

$$\frac{s \text{ is } f_1\text{-closed}_{g_1}^{g_2} \quad s \text{ is } f_2\text{-closed}_{g_1}^{g_2}}{s \text{ is } (f_1 \cup f_2)\text{-closed}_{g_1}^{g_2}}$$ (ii) Closure under union

$$\frac{s_1 \text{ is } f\text{-closed}_{g_1}^{g_2} \quad s_2 \text{ is } f\text{-closed}_{g_1}^{g_2}}{(s_1 \cap s_2) \text{ is } f\text{-closed}_{g_1}^{g_2}}$$ (iii) Closure under intersection

$$\frac{s \text{ is } f\text{-closed}_{g_1}^{g_2}}{\overline{s} \text{ is } \overline{f}\text{-closed}_{\overline{g_1}}^{\overline{g_2}}}$$ (iv) Closure under transposition

$$\frac{f \text{ symmetric} \quad g_1, g_2 \text{ symmetric} \quad s \text{ is } f\text{-closed}_{g_1}^{g_2}}{\overleftrightarrow{s} \text{ is } f\text{-closed}_{g_1}^{g_2}}$$ (v) Closure under symmetrization

Figure 5.1: Closure properties of conditional closedness

f-closed$_{g_1}^{g_2}$ and g_1, g_2 to be symmetric, we get that $\overleftrightarrow{s}$ is f-closed$_{g_1}^{g_2}$ (by Figure 5.1 (iii), (v)). Then we can separately show $g_1(\nu \overleftrightarrow{s}) \subseteq \nu \overleftrightarrow{s}$ and $\nu \overleftrightarrow{s} \subseteq g_2(\nu \overleftrightarrow{s})$ and get the closure property.

If components are "merged" as in Figure 5.1 (ii), (iii), then their minimum and maximum condition might not be the same at first. This is no problem though, as one can always increase the minimum and decrease the maximum condition of a conditional closedness statement by Figure 5.1 (i). Hence the conditions of different components can be aligned so that rules (ii) and (iii) of Figure 5.1 can be properly applied.

5.2.2 Discussion

Shifting from compatibility to conditional closedness via adding the two side conditions expressing minimum and maximum conditions, we cannot use conditional closedness as a soundness criterion for up-to techniques anymore. It would be nice to have a criterion for soundness (Definition 4.1) with regard to greatest fixed-points νs which fulfill certain conditions. Consider conditional companions defined as follows.

$$ {}_s c_{g_1}^{g_2} := \bigcup \{ f \mid s \text{ is } f\text{-closed}_{g_1}^{g_2} \ \wedge \ f \text{ monotone} \}$$

Now one would like to obtain Lemma 4.4 for companions and greatest fixed-points which abide by minimum and maximum conditions. The proof of Lemma 4.4 in [12] is based upon the property $c_s \circ c_s \subseteq c_s$. In our case of conditional companions, this corresponds to the property ${}_s c_{g_1}^{g_2} \circ {}_s c_{g_1}^{g_2} \subseteq {}_s c_{g_1}^{g_2}$. At this point we would need conditional closedness to be closed under function composition. Unfortunately we have not found such an appropriate closure property. We think though that a further

investigation of a potential “conditional compatibility” could be rewarding and mark this as future work.

Chapter 6

Substitutivity of IO Bisimilarity

In the following, we show substitutivity of IO bisimilarity. Substitutivity essentially states that the instantiation of free variables of two processes preserves their bisimilarity. Being a closure property, we recall from Definition 5.1 that the substitution closure is defined as

$$f_{subst}(\mathcal{R}) := \{(A[\sigma], B[\sigma]) \mid (A, B) \in \mathcal{R},\ \sigma \text{ substitution}\}$$

We will show that $\nu\overleftrightarrow{s_{io}}$ is closed under f_{subst}, i.e. $f_{subst}(\nu\overleftrightarrow{s_{io}}) \subseteq \nu\overleftrightarrow{s_{io}}$.

Having two criteria for closedness at hand (compatibility and conditional closedness), we show substitutivity using conditional closedness. As already hinted at in Section 5.2, we can show greatest fixed-points of functionals including s_{ho_out} only to be closed under f_{subst} if they are at least reflexive and relate at most processes with the same unguarded variables. As IO bisimilarity fulfills these two conditions, conditional closedness lets us prove this result in a compositional manner. As a means of requiring the same unguarded variables we use the fact that IO bisimilarity is a $\overleftrightarrow{s_{var\_cxt}}$-bisimulation.

Being an $\overleftrightarrow{s_{var\_cxt}}$-bisimulation is clearly a post-fixed-point property, i.e. s_{var_cxt} can be given as a maximum condition. Reflexivity can be expressed through a pre-fixed-point property: Because a relation $\mathcal{R}$ is reflexive iff $1 \subseteq \mathcal{R}$, the constant function $\boldsymbol{f_{refl}} := \hat{1}$ can be given as a minimum condition.

Theorem 6.1 $f_{subst}(\nu\overleftrightarrow{s_{io}}) \subseteq \nu\overleftrightarrow{s_{io}}$

Proof By the closure properties of conditional closedness (Figure 5.1) we have the following derivation:

$$\frac{\begin{array}{lll} s_{ho\_out} & f_{subst}\text{-closed}^{s_{var\_cxt}}_{f_{refl}} & \\ s_{ho\_in} & f_{subst}\text{-closed}^{s_{var\_cxt}}_{\bot} & f_{subst} \text{ symmetric} \\ s_{var\_cxt} & f_{subst}\text{-closed}^{\top}_{\bot} & f_{refl} \text{ symmetric} \end{array}}{\overleftrightarrow{s_{io}}\ f_{subst}\text{-closed}^{\overleftrightarrow{s_{var\_cxt}}}_{f_{refl}}}$$

We show the conditional closedness proofs for the components s_{ho_out}, s_{ho_in} and s_{var_cxt} in Section 6.3. Symmetry of f_{subst} and f_{refl} is trivial. By Lemma 5.4, it

remains to show the minimum condition for IO bisimilarity, i.e. $f_{refl}(\nu\overleftrightarrow{s_{io}}) \subseteq \nu\overleftrightarrow{s_{io}}$. We prove this in Lemma 6.3. ■

With regard to the implementation in Coq, the proof is not completely automatic. This is due to the fact that specifying common minimum and maximum conditions, as well as showing that they are appropriate, is done manually. In principle though this could be solved by automation.

6.1 Reflexivity of IO Bisimilarity

Having to show reflexivity of a compositional definition of bisimilarity, we find ourselves in a similar situation as before: Our goal is to prove reflexivity separately for the components (in our case, for s_{ho_out}, s_{ho_in} and s_{var_cxt}) and then merge them together. We can do so by using the notion of compatibility again. The following equivalence[1] holds.

$$f_{refl} \text{ is } s\text{-compatible} \quad \Leftrightarrow \quad 1 \subseteq s(1)$$

We show $1 \subseteq s(1)$ for each component.

Proposition 6.2 $1 \subseteq s_{ho\_out}(1)$ *and* $1 \subseteq s_{ho\_in}(1)$ *and* $1 \subseteq s_{var\_cxt}(1)$.

Lemma 6.3 $1 \subseteq \nu\overleftrightarrow{s_{io}}$

Proof By the closure properties of compatibility (Figure 4.1) with Proposition 6.2 and symmetry of f_{refl}, we have f_{refl} is $\overleftrightarrow{s_{io}}$-compatible. By Lemma 5.2, this implies reflexivity of νs. ■

6.2 Transitions of Substituted Processes

To prove conditional compatibility of the different components in Section 6.3, we need to analyze transitions of substituted processes, e.g. $A[\sigma] \xrightarrow{\overline{a}\langle B\rangle} C$. In this case there are exactly two possibilities of how the output transition can happen:

1. Process A itself can make an output transition on channel a. Hence $A[\sigma]$ can also make a transition on channel a.
2. Process A has an unguarded occurrence of variable x and the process $\sigma(x)$ can output B on a. Thus $A[\sigma]$ can output B on a, too.

Deriving a transition of a substituted process as in (1) is described by the substitutivity property (Section 2.3.1). Deriving a transition of a substituted process as in (2) makes use of variable context transitions and is described in the following[2]:

Lemma 6.4 (Propagation via context)

$$\frac{A \xrightarrow{x} C \quad \sigma(x) \xrightarrow{\overline{a}\langle B\rangle} E}{A[\sigma] \xrightarrow{\overline{a}\langle B\rangle} C[E :: \sigma]} \qquad \frac{A \xrightarrow{x} C \quad \sigma(x) \xrightarrow{a} E}{A[\sigma] \xrightarrow{a} C[E :: (\sigma \circ \uparrow)]} \qquad \frac{A \xrightarrow{x} C \quad \sigma(x) \xrightarrow{y} E}{A[\sigma] \xrightarrow{y} C[E :: (\sigma \circ \uparrow)]}$$

[1] The criterion "$1 \subseteq s(1)$" is introduced in [11] as part of "monoid preservation".

[2] Propagation via context also holds for τ-transitions, but is not stated here as it is not used.

We can show that output and input transitions of substituted processes can only be obtained either by substitutivity (Lemma 2.1) or by propagation via context (Lemma 6.4). We can further show that variable context transitions can only be obtained by substitutivity Lemma 2.1. Therefore, we have an inversion lemma to analyze such transitions and distinguish between the two derivations.

Lemma 6.5 (Inversion of transitions of substituted processes)

$$A[\sigma] \xrightarrow{\overline{a}\langle B\rangle} D \quad \Leftrightarrow \quad \begin{array}{l} \exists\, B', D'.\ A \xrightarrow{\overline{a}\langle B'\rangle} D' \wedge B = B'[\sigma] \wedge D = D'[\sigma] \\ \textbf{\textit{or}} \\ \exists\, x, C, E.\ A \xrightarrow{x} C \wedge \sigma(x) \xrightarrow{\overline{a}\langle B\rangle} E \wedge D = C[E :: \sigma] \end{array}$$

$$A[\sigma] \xrightarrow{a} B \quad \Leftrightarrow \quad \begin{array}{l} \exists\, B'.\ A \xrightarrow{a} B' \wedge B = B'[\Uparrow \sigma] \\ \textbf{\textit{or}} \\ \exists\, x, C, E.\ A \xrightarrow{x} C \wedge \sigma(x) \xrightarrow{a} E \wedge B = C[E :: (\sigma \circ \uparrow)] \end{array}$$

$$A[\sigma] \xrightarrow{x} C \quad \Leftrightarrow \quad \exists\, y, C'.\ A \xrightarrow{y} C' \wedge \sigma(y) \xrightarrow{x} E \wedge C = C'[E :: (\sigma \circ \uparrow)]$$

Proof $\Leftarrow$: By Lemma 2.1 (substitutivity) and Lemma 6.4 (propagation via context).
$\Rightarrow$: By induction on A. ∎

6.3 Conditional Closedness of Components

In the following we present how the conditional closedness proofs for the individual components can be done.

Lemma 6.6

$$s_{ho\_out} \text{ is } f_{subst}\text{-closed}^{s_{var\_cxt}}_{f_{refl}} \qquad s_{ho\_in} \text{ is } f_{subst}\text{-closed}^{s_{var\_cxt}}_{\perp} \qquad s_{var\_cxt} \text{ is } f_{subst}\text{-closed}^{\top}_{\perp}$$

Proof We show only conditional closedness of s_{ho_out}, the proofs for s_{ho_in} and s_{var_cxt} are similar.

Assume a relation $\mathcal{R}$ with $f_{refl}(\mathcal{R}) \subseteq \mathcal{R}$, i.e. $\mathcal{R}$ is reflexive, with $\mathcal{R} \subseteq s_{var\_cxt}(\mathcal{R})$ and with $\mathcal{R} \subseteq s_{ho\_out}(\mathcal{R})$. We have to show $f_{subst}(\mathcal{R}) \subseteq s_{ho\_out}(f_{subst}(\mathcal{R}))$.

Let $(A, B) \in \mathcal{R}$ with a transition $A[\sigma] \xrightarrow{\overline{a}\langle A'\rangle} A''$. We have to show the existence of B', B'' s.t. $B[\sigma] \xrightarrow{\overline{a}\langle B'\rangle} B''$ with $(A', B') \in f_{subst}(\mathcal{R})$ and $(A'', B'') \in f_{subst}(\mathcal{R})$. Lemma 6.5 lets us distinguish between two cases:

1. $A \xrightarrow{\overline{a}\langle \hat{A}'\rangle} \hat{A}''$ with $A' = \hat{A}'[\sigma]$ and $A'' = \hat{A}''[\sigma]$.

   By $\mathcal{R} \subseteq s_{ho\_out}(\mathcal{R})$ there exists $\hat{B}'$, $\hat{B}''$ s.t. $B \xrightarrow{\overline{a}\langle \hat{B}'\rangle} \hat{B}''$ with $(\hat{A}', \hat{B}') \in \mathcal{R}$ and $(\hat{A}'', \hat{B}'') \in \mathcal{R}$.

   Finally, substitutivity (Lemma 2.1) yields $B[\sigma] \xrightarrow{\overline{a}\langle \hat{B}'[\sigma]\rangle} \hat{B}''[\sigma]$. Our claim follows by $(\hat{A}'[\sigma], \hat{B}'[\sigma]) \in f_{subst}(\mathcal{R})$ and $(\hat{A}''[\sigma], \hat{B}''[\sigma]) \in f_{subst}(\mathcal{R})$.

2. $A \xrightarrow{x} C_A$ with $\sigma(x) \xrightarrow{\overline{a}\langle A' \rangle} E$ and $A'' = C_A[E :: \sigma]$.

   By $\mathcal{R} \subseteq s_{var\_cxt}(\mathcal{R})$ there exists C_B s.t. $B \xrightarrow{x} C_B$ with $(C_A, C_B) \in \mathcal{R}$.

   By propagation via context(Lemma 6.4), we obtain $B[\sigma] \xrightarrow{\overline{a}\langle A' \rangle} C_B[E :: \sigma]$. It remains to show:

   (a) $(A', A') \in f_{subst}(\mathcal{R})$. This holds by extensiveness of f_{subst} ($id \subseteq f_{subst}$) and reflexivity of $\mathcal{R}$.

   (b) $(C_A[E :: \sigma], C_B[E :: \sigma]) \in f_{subst}(\mathcal{R})$. This holds by $(C_A, C_B) \in \mathcal{R}$. ∎

6.4 Injective Renamings

To show substitutivity of IO bisimilarity we relied on it being reflexive and a variable context bisimulation. We will now consider a subset of substititions, namely decidable injective renamings. A **decidable injective renaming** is an injective renaming for which one can decide for every variable if it is in the image of the renaming or not. We find this requirement to be fulfilled by all injective renamings we are working with[3]. As decidable injective renamings form a subset of substitutions, closedness under substitions implies closedness under decidable injective renamings. It turns out that closedness under decidable injective renamings does not impose any conditions on an enclosing bisimilarity as substitutivity does and that we can prove compatibility for the corresponding closure function. In doing so, we obtain a sound up-to technique and a closedness property of IO bisimilarity at the same time. We define the **decidable injective renaming closure** as follows:

$$f_{dec\_inj\_ren}(\mathcal{R}) := \{(A[\xi], B[\xi]) \mid (A, B) \in \mathcal{R},\ \xi \text{ decidable injective renaming}\}$$

We show compatibility for every component.

Lemma 6.7 *$f_{dec\_inj\_ren}$ is s_{ho_out}-, s_{ho_in}- and s_{var_cxt}-compatible*

Proof The proofs for the components are similar, exemplarily we show that $f_{dec\_inj\_ren}$ is s_{ho_out}-compatible:

Let ξ be a decidable injective renaming and $\mathcal{R}, \mathcal{S}$ two relations with $\mathcal{R} \subseteq s_{ho\_out}(\mathcal{S})$. We have to show $f_{dec\_inj\_ren}(\mathcal{R}) \subseteq s_{ho\_out}(f_{dec\_inj\_ren}(\mathcal{S}))$.

Let $(A, B) \in \mathcal{R}$ with a transition $A[\xi] \xrightarrow{\overline{a}\langle A' \rangle} A''$. We have to show the existence of B', B'' s.t. $B[\xi] \xrightarrow{\overline{a}\langle B' \rangle} B''$ with $(A', B') \in f_{dec\_inj\_ren}(\mathcal{S})$ and $(A'', B'') \in f_{dec\_inj\_ren}(\mathcal{S})$.

By preservation of renaming(Lemma 2.2) we have the existence of $\hat{A}'$, $\hat{A}''$ s.t. $A \xrightarrow{\overline{a}\langle \hat{A}' \rangle} \hat{A}''$ with $A' = \hat{A}'[\xi]$ and $A'' = \hat{A}''[\xi]$. By assumption, there exist also $\hat{B}'$, $\hat{B}''$ s.t. $B \xrightarrow{\overline{a}\langle \hat{B}' \rangle} \hat{B}''$ with $(\hat{A}', \hat{B}') \in \mathcal{S}$ and $(\hat{A}'', \hat{B}'') \in \mathcal{S}$. We pick $B' = \hat{B}'[\xi]$ and

[3]The requirement of being decidable is not necessary in this case. We have included it because it is needed for s_{var_multi}-compatibility (Chapter 8) which can then be automatically composed with the compatibility results of this section.

$B'' = \hat{B''}[\xi]$. By substitutivity (Lemma 2.1) we have $B[\xi] \xrightarrow{\overline{a}\langle B' \rangle} B''$ and by definition we have $(A', B') \in f_{dec\_inj\_ren}(\mathcal{S})$ and $(A'', B'') \in f_{dec\_inj\_ren}(\mathcal{S})$. ■

Lemma 6.8 *$f_{dec\_inj\_ren}$ is $\overleftrightarrow{s_{io}}$-compatible*

Proof By the closure properties of compatibility (Fig. 4.1) with Lemma 6.7 and symmetry of $f_{dec\_inj\_ren}$. ■

Chapter 7

Congruence of IO Bisimilarity

In the following, we show that IO bisimilarity is a congruence. Intuitively, this means that filling the hole of a context with bisimilar processes results again in bisimilar processes. Formally, we can express congruence as a closure property using the following **context closure**.

$$f_{cxt}(\mathcal{R}) := \{(C[\![A]\!], C[\![B]\!]) \mid (A, B) \in \mathcal{R}, \quad C \ \textit{context}\}$$

Note that we are using a different instantiation operation $[\![\cdot]\!]$ instead of $[\cdot]$. We call $[\![\cdot]\!]$ a **capturing instantiation**, because it allows the substituted processes to be captured by existing binders. The only difference between the definition of $[\cdot]$ and $[\![\cdot]\!]$ lies in the following case.

$$(a.A)\,[\sigma] := a.(A[\Uparrow \sigma])$$
$$(a.A)[\![\sigma]\!] := a.(A[\![0 :: \sigma]\!])$$

To show the different impact with regard to capturing, consider the following example: With $[\![\cdot]\!]$, we cannot instantiate the context $a.1$ (with 1 being the context hole) in such a way that we get as a filled context $a.0$. While $(a.1)[0] = a.1$, we have $(a.1)[\![0]\!] = a.0$.

We will have as a final result that $\nu\overleftrightarrow{s_{io}}$ is closed under f_{cxt}. For the proof to go through, we show a more general closure property, based on the following definition of a **multi-context closure** (we interpret the free variables of A as context holes).

$$f_{multi\_cxt}(\mathcal{R}) := \{(A[\![\sigma]\!], A[\![\tau]\!]) \mid \ \sigma, \tau \ \textit{substitutions} \ \wedge \ \forall x \in \mathbb{N}.\ (\sigma\ x, \tau\ x) \in \mathcal{R}\}$$

In [6], congruence of IO bisimilarity is given in three parts, as congruence for output processes, input-prefixed processes and parallel composition. We split up the congruence proof in a similar way, using the following three closures.

$$f_{send}(\mathcal{R}) := \{(\overline{a}\langle A\rangle, \overline{a}\langle B\rangle) \ \mid \ (A, B) \in \mathcal{R}\}$$
$$f_{receive}(\mathcal{R}) := \{(a.A, a.B) \ \mid \ (A, B) \in \mathcal{R}\}$$
$$f_{par}(\mathcal{R}) := \{(A_1 \parallel A_2, B_1 \parallel B_2) \ \mid \ (A_1, A_2) \in \mathcal{R} \ \wedge \ (B_1, B_2) \in \mathcal{R}\}$$

Theorem 7.1 $f_{multi\_cxt}(\nu \overleftrightarrow{s_{io}}) \subseteq \nu \overleftrightarrow{s_{io}}$

Proof Assume a process A. We have to show that for every pair of substitutions σ, τ with $\forall\, x \in \mathbb{N}.\ \sigma(x) \sim_{io} \tau(x)$, we also have $A[\![\sigma]\!] \sim_{io} A[\![\tau]\!]$. This holds by induction over A using closedness of f_{send}, $f_{receive}$ and f_{par}, which we show next. ∎

It remains to show $f_{send}(\nu \overleftrightarrow{s_{io}}) \subseteq \nu \overleftrightarrow{s_{io}}$, $f_{receive}(\nu \overleftrightarrow{s_{io}}) \subseteq \nu \overleftrightarrow{s_{io}}$ and $f_{par}(\nu \overleftrightarrow{s_{io}}) \subseteq \nu \overleftrightarrow{s_{io}}$. As in Chapter 6, we use conditional closedness with appropriate conditions, treating the different components of $\nu \overleftrightarrow{s_{io}}$ separately. For the conditional closedness proofs to go through, we show conditional closedness of the increasing versions of the three closures, yielding slightly stronger statements. The **increasing version $f \cup id$ of a function f** is denoted as $\mathring{f}$.

For every closure $\mathring{f_{send}}$, $\mathring{f_{receive}}$, $\mathring{f_{par}}$, this leaves us with three proofs of conditional closedness, one for each of the components s_{ho_out}, s_{ho_in}, s_{var_cxt}. We require f_{refl} and $f_{dec\_inj\_ren}$ as conditions. The following derivations consist of mechanic applications of closure rules of conditional closedness (Figure 5.1).

$$\frac{\begin{array}{ll} f_{refl} & \textit{symmetric} \\ \mathring{f_{send}} & \textit{symmetric} \\ s_{ho\_out} & \mathring{f_{send}}\text{-}\textit{closed}^{\top}_{f_{refl}} \\ s_{ho\_in} & \mathring{f_{send}}\text{-}\textit{closed}^{\top}_{\bot} \\ s_{var\_cxt} & \mathring{f_{send}}\text{-}\textit{closed}^{\top}_{\bot} \end{array}}{\overleftrightarrow{s_{io}}\ \mathring{f_{send}}\text{-}\textit{closed}^{\top}_{f_{refl}}} \qquad \frac{\begin{array}{ll} \mathring{f_{receive}} & \textit{symmetric} \\ s_{ho\_out} & \mathring{f_{receive}}\text{-}\textit{closed}^{\top}_{\bot} \\ s_{ho\_in} & \mathring{f_{receive}}\text{-}\textit{closed}^{\top}_{\bot} \\ s_{var\_cxt} & \mathring{f_{receive}}\text{-}\textit{closed}^{\top}_{\bot} \end{array}}{\overleftrightarrow{s_{io}}\ \mathring{f_{receive}}\text{-}\textit{closed}^{\top}_{\bot}} \qquad \frac{\begin{array}{ll} f_{dec\_inj\_ren} & \textit{symmetric} \\ \mathring{f_{par}} & \textit{symmetric} \\ s_{ho\_out} & \mathring{f_{par}}\text{-}\textit{closed}^{\top}_{\bot} \\ s_{ho\_in} & \mathring{f_{par}}\text{-}\textit{closed}^{\top}_{f_{dec\_inj\_ren}} \\ s_{var\_cxt} & \mathring{f_{par}}\text{-}\textit{closed}^{\top}_{f_{dec\_inj\_ren}} \end{array}}{\overleftrightarrow{s_{io}}\ \mathring{f_{par}}\text{-}\textit{closed}^{\top}_{f_{dec\_inj\_ren}}}$$

Symmetry of f_{refl}, $f_{dec\_inj\_ren}$, $\mathring{f_{send}}$, $\mathring{f_{receive}}$ and $\mathring{f_{par}}$ are trivial. In the following we present how the proofs for the individual components can be done. We demonstrate the proof of conditional closedness of $\mathring{f_{par}}$ for s_{ho_in}, the other proofs are similar.

Lemma 7.2 s_{ho_in} *is* $\mathring{f_{par}}$*-closed*$^{\top}_{f_{dec\_inj\_ren}}$

Proof Assume a relation $\mathcal{R}$ with $f_{dec\_inj\_ren}(\mathcal{R}) \subseteq \mathcal{R}$, i.e. $\mathcal{R}$ is closed under decidable injective renamings, and with $\mathcal{R} \subseteq s_{ho\_in}(\mathcal{R})$. We have to show $\mathring{f_{par}}(\mathcal{R}) \subseteq s_{ho\_in}(\mathring{f_{par}}(\mathcal{R}))$. As we are dealing with the decreasing version of f_{par}, this consists of two parts: First, showing $\mathcal{R} \subseteq s_{ho\_in}(\mathring{f_{par}}(\mathcal{R}))$, which follows by assumption. Second, it remains to show $f_{par}(\mathcal{R}) \subseteq s_{ho\_in}(\mathring{f_{par}}(\mathcal{R}))$.

Let $(A_1, B_1) \in \mathcal{R}$ and $(A_2, B_2) \in \mathcal{R}$ with a transition $A_1 \parallel A_2 \xrightarrow{a} A$. We have to show the existence of B s.t. $B_1 \parallel B_2 \xrightarrow{a} B$ with $(A, B) \in \mathring{f_{par}}(\mathcal{R})$. We do a case analysis on the transition which is either obtained by InParL or by InParR:

1. $A_1 \parallel A_2 \xrightarrow{a} A' \parallel A_2[\uparrow]$ with $A_1 \xrightarrow{a} A'$. By assumption there is also a transition $B_1 \xrightarrow{a} B'$ with $(A', B') \in \mathcal{S}$. We pick $B = B' \parallel B_2[\uparrow]$ and have $B_1 \parallel B_2 \xrightarrow{a} B' \parallel B_2[\uparrow]$ by InParL. It remains to show $(A' \parallel A_2[\uparrow], B' \parallel B_2[\uparrow]) \in \mathring{f_{par}}(\mathcal{R})$. By assumption and definition of f_{par} it suffices to show $(A_2[\uparrow], B_2[\uparrow]) \in \mathcal{R}$. We have this because $\mathcal{R}$ is closed under decidable injective renamings.
2. $A_1 \parallel A_2 \xrightarrow{a} A_1[\uparrow] \parallel A'$ with $A_2 \xrightarrow{a} A'$. *Analog.* ∎

Chapter 8

Handling Unguarded Variables

Besides being useful for the analysis of processes through contexts, variable context transitions are used through the simulation functional s_{var_cxt} to check that the unguarded variable occurrences of two bisimilar processes coincide. Our way of performing this check for IO bisimilarity through s_{var_cxt} differs from other approaches which are present in the literature [6, 7]. In this chapter, we compare existing ways for the treatment of unguarded variable occurrences and prove that for IO bisimilarity, three of them are interchangeable. This justifies using variable context bisimulations for the definition of IO bisimilarity.

The following approaches are present in the literature:

1. In [6], checking coincidence of unguarded variable occurrences is based upon a definition of structural congruence ($\equiv$)[1]. The corresponding functional s is the following:

   $(A, B) \in s(\mathcal{R})$ iff for each x, A' with $A \equiv x \parallel A'$ there is a B' s.t. $B \equiv x \parallel B'$ with $(A', B') \in \mathcal{R}$.

2. In [7], *variable removal simulations*[2] are used, based upon variable removal transitions. **Variable removal transitions** ($\stackrel{x}{\Longrightarrow}$) replace unguarded variable occurrences by $\emptyset$ and can be defined in terms of variable context transitions:

   $$\frac{A \stackrel{x}{\rightarrow} C}{A \stackrel{x}{\Longrightarrow} C[\emptyset]}$$

   Variable removal simulations are defined through the following functional: $\boldsymbol{s_{var\_rem}}(\mathcal{R})$ contains a pair (A, B) iff each transition $A \stackrel{x}{\Longrightarrow} A'$ implies that there is a B' s.t. $B \stackrel{x}{\Longrightarrow} B'$ with $(A', B') \in \mathcal{R}$.

We chose not to use the approach of [6], because we want to avoid dealing with structural congruence. Our variable context transitions can be seen as an extension of variable removal transitions [7]. Besides the advantage gained for the substitutivity proof, there is also a downside to our simulation functional s_{var_cxt}: Context variable transitions do not reduce the size of processes and thereby disable a direct decision procedure for bisimilarity which is based on a terminating character of the functional.

[1] To be precise, the simulation is part of the definition of "open simulation" [6].

[2] To be precise, the simulations are coined variable simulations [7] but to avoid confusion, we emphasize its removal character.

8.1 Termination Behavior

The definitions of IO bisimilarity through structural congruence [6] and through s_{var_rem} [7] stand out of all definitions of bisimilarity because one can give a direct decision procedure for it, i.e. every two processes A and B can be proven either bisimilar or not bisimilar. This is due to the terminating character of each component s_{ho_out}, s_{ho_in} and s_{var_multi}: One can give a **size function** for processes in a way that each transition produces strictly smaller processes [6].

$$\begin{aligned} size(\emptyset) &= 0 \\ size(x) &= 1 \\ size(\overline{a}\langle A \rangle) &= 1 + size(A) \\ size(a.A) &= 1 + size(A) \\ size(A \parallel B) &= size(A) + size(B) \end{aligned}$$

1. For each of the finite number of *output* transitions, the emitted and target processes are strictly smaller than the source process. This requires only a finite number of checks.

2. For each of the finite number of *input* transitions, the target proces is strictly smaller than the source process. Noticeable about s_{ho_in} is that no variables are instantiated when checking that the target processes A' and B' of two input transitions are in $\mathcal{R}$. This could lead to larger processes than before, due to the substitution of the 0 variables in the target processes. In our setting though, only a finite number of checks is required.

3. For checking unguarded variable coincidence, the three methods (via structural congruence [6], variable removal simulations through s_{var_rem} [7], variable context simulations through s_{var_cxt}) differ in their termination behavior. For the structural congruence approach [6] and s_{var_rem} approach [7], checks only need to be performed on strictly smaller processes as an unguarded variable is either completely removed or replaced by $\emptyset$ after each check. Because there is only a finite number of unguarded variables in a process, one needs to do only a finite number of checks.

   Checking simulations defined through s_{var_cxt} may lead to an infinite number of checks, because variable context transitions do not decrease the size of a process as variables are only renamed. An example of a process causing infinitely many checks is the following flip-flop example:

$$(0 \parallel 1 \parallel 3) \xrightarrow{1} (1 \parallel 0 \parallel 4) \xrightarrow{1} (0 \parallel 1 \parallel 5) \xrightarrow{1} ...$$

For the two approaches in the literature, there exist direct decision procedures based on the finiteness of checks to be performed [6, 7] and Maksimović et al. have formalized it in Coq [7]. For our definition of IO bisimilarity using s_{var_cxt}, we cannot carry over the direct decision procedure from the literature.

To show that our definition describes the same IO bisimilarity as in the literature, we prove in this chapter that our definition of IO bisimilarity using s_{var_cxt} coincides with the definition using s_{var_rem}. We have not yet formalized the decidability result itself for IO bisimilarity on our own due to a lack of time.

8.2 Coincidence of IO Bisimilarity Variants

We do not directly show coincidence of the definition of IO bisimilarity based on s_{var_cxt} and the definition based on s_{var_rem}. The reason for this is that we would have to rely on substitutivity of both greatest fixed-points, but only have this property for IO bisimilarity defined via s_{var_cxt} (Theorem 6.1).

We introduce an auxiliary definition which ensures coincidence of unguarded variables, serving as a bridge. Taking a straightforward way, we require that the multisets of unguarded variables of two processes are equivalent. We use the notation $\boldsymbol{V(A)}$ for the **multiset of unguarded variables** which are contained in a process A and $\boldsymbol{\mathrm{mul}_x(V(A))}$ for the **multiplicity of a variable** x within $V(A)$. In Coq, we have implemented multisets as unordered lists over natural numbers (see also Chapter 9).

The checks are required through a constant functional which we call **variable multiset** functional.

Definition 8.1 $(A, B) \in \boldsymbol{s_{var\_multi}}(\mathcal{R})$ *iff* $V(A) \subseteq V(B)$.

We have now three definitions of IO bisimilarity, each of them dealing with unguarded variables in a different way.

$$\begin{aligned} s_{io} = \boldsymbol{s_{io\_cxt}} &:= s_{ho\_out} \cap s_{ho\_in} \cap s_{var\_cxt} \\ \boldsymbol{s_{io\_rem}} &:= s_{ho\_out} \cap s_{ho\_in} \cap s_{var\_rem} \\ \boldsymbol{s_{io\_multi}} &:= s_{ho\_out} \cap s_{ho\_in} \cap s_{var\_multi} \end{aligned}$$

We show $\nu\overleftrightarrow{s_{io\_cxt}} = \nu\overleftrightarrow{s_{io\_rem}} = \nu\overleftrightarrow{s_{io\_multi}}$ in the following circular way.

$$\begin{aligned} \nu\overleftrightarrow{s_{io\_cxt}} &\subseteq \nu\overleftrightarrow{s_{io\_rem}} \\ \nu\overleftrightarrow{s_{io\_rem}} &\subseteq \nu\overleftrightarrow{s_{io\_multi}} \\ \nu\overleftrightarrow{s_{io\_multi}} &\subseteq \nu\overleftrightarrow{s_{io\_cxt}} \end{aligned}$$

Because the greatest fixed-points are symmetric and differ only in the way they deal with unguarded variables, it suffices to show that their treatment of unguarded variables can be simulated.

$$\begin{aligned} \nu\overleftrightarrow{s_{io\_cxt}} &\subseteq s_{var\_rem} \quad (\nu\overleftrightarrow{s_{io\_cxt}}) \\ \nu\overleftrightarrow{s_{io\_rem}} &\subseteq s_{var\_multi} (\nu\overleftrightarrow{s_{io\_rem}}) \\ \nu\overleftrightarrow{s_{io\_multi}} &\subseteq s_{var\_cxt} \quad (\nu\overleftrightarrow{s_{io\_multi}}) \end{aligned}$$

It turns out that proving the first two inclusions is straightforward through elegant proofs. The third proof needs more effort, as we will explain later. For all three inclusions, we rely on strong connections between the three concepts.

Proposition 8.2

$$\begin{aligned} A \overset{x}{\Longrightarrow} A' &\quad\Leftrightarrow\quad \exists\, C.\ A \xrightarrow{x} C \wedge A' = C[\emptyset] && \text{(i)} \\ \mathrm{mul}_x(V(A)) > 0 &\quad\Leftrightarrow\quad \exists\, A'.\ A \overset{x}{\Longrightarrow} A' && \text{(ii)} \\ \exists\, C.\ A \xrightarrow{x} C &\quad\Leftrightarrow\quad \mathrm{mul}_x(V(A)) > 0 && \text{(iii)} \\ \\ A \overset{x}{\Longrightarrow} A' &\quad\Rightarrow\quad \mathrm{mul}_x(V(A)) = \mathrm{mul}_x(V(A')) + 1 && \text{(iv)} \end{aligned}$$

We present now proofs of the first two inclusions.

Lemma 8.3 $\nu\overleftarrow{s_{io\_cxt}} \subseteq s_{io\_rem}(\nu\overleftarrow{s_{io\_cxt}})$

Proof Let $(A, B) \in \nu\overleftarrow{s_{io\_cxt}}$ with $A \stackrel{x}{\Longrightarrow} A'$. We have to show the existence of B' s.t. $B \stackrel{x}{\Longrightarrow} B'$ with $(A', B') \in \nu\overleftarrow{s_{io\_cxt}}$.

By Proposition 8.2 (i), there is a context C_A with $A \stackrel{x}{\rightarrow} C_A$ and $A' = C_A[\emptyset]$. Because A and B are bisimilar, there exists a context C_B with $B \stackrel{x}{\rightarrow} C_B$ and $(C_A, C_B) \in \nu\overleftarrow{s_{io\_cxt}}$. By definition of variable removal transitions, we have $B \stackrel{x}{\Longrightarrow} C_B[\emptyset]$. It remains to show $(C_A[\emptyset], C_B[\emptyset]) \in \nu\overleftarrow{s_{io\_cxt}}$, which we have by substitutivity (Theorem 6.1). ∎

Lemma 8.4 $\nu\overleftarrow{s_{io\_rem}} \subseteq s_{io\_multi}(\nu\overleftarrow{s_{io\_rem}})$

Proof Let $(A, B) \in \nu\overleftarrow{s_{io\_rem}}$. We have to show $V(A) \subseteq V(B)$. Consider a variable n. We have to show $\mathrm{mul}_x(V(A)) \leq \mathrm{mul}_x(V(B))$. We prove this by complete induction over $\mathrm{mul}_x(V(A)) + \mathrm{mul}_x(V(B))$ and distinguish between two cases:

1. $\mathrm{mul}_x(V(A)) = 0$. *Trivial.*
2. $\mathrm{mul}_x(V(A)) > 0$. By Proposition 8.2 (ii), there exists a process A' s.t. $A \stackrel{x}{\Longrightarrow} A'$. Because A and B are bisimilar, there exists also a process B' with $B \stackrel{x}{\Longrightarrow} B'$ and $(A', B') \in \nu\overleftarrow{s_{io\_rem}}$. By Proposition 8.2 (ii), we have $\mathrm{mul}_x(V(B)) > 0$. Using Proposition 8.2 (iv), it suffices to show $\mathrm{mul}_x(V(A')) \leq \mathrm{mul}_x(V(B'))$. We have this by induction because $\mathrm{mul}_x(V(A')) + \mathrm{mul}_x(V(B')) \leq \mathrm{mul}_x(V(A)) + \mathrm{mul}_x(V(B))$. ∎

The third inclusion $\nu\overleftarrow{s_{io\_multi}} \subseteq s_{var\_cxt}(\nu\overleftarrow{s_{io\_multi}})$ requires more effort, because the constant functional s_{var_multi} does not relate any potential target processes of A and B. We apply a trick to use conditional closedness to tackle the problem: Because conditional closedness only gives us closure properties of the form $f(\nu s) \subseteq \nu s$, we "invert" the functional s_{var_cxt} in a certain way and receive its counterpart which we call f_{reach}. Showing $f_{reach}(\nu\overleftarrow{s_{io\_multi}}) \subseteq \nu\overleftarrow{s_{io\_multi}}$ will help us showing our desired goal, $\nu\overleftarrow{s_{io\_multi}} \subseteq s_{var\_cxt}(\nu\overleftarrow{s_{io\_multi}})$. We define $\boldsymbol{f_{reach}}$ as follows, together with the **closure under bijective renamings**.

$$f_{reach}(\mathcal{R}) := \{(A', B') \mid \exists\, A, B, x.\ (A, B) \in \mathcal{R} \wedge A \stackrel{x}{\rightarrow} A' \wedge B \stackrel{x}{\rightarrow} B'\}$$

$$f_{bij\_ren}(\mathcal{R}) := \{(A[\xi], B[\xi]) \mid (A, B) \in \mathcal{R},\ \xi \textit{ bijective renaming}\}$$

Before we come to the proof of the third inclusion in Lemma 8.8, we state two propositions. First, we state closedness of $\nu\overleftarrow{s_{io\_multi}}$ under decidable injective renamings. Note that for the s_{ho_out} and s_{ho_in} components, we have presented proofs for this in Lemma 6.7. Second, we state the three main components for the inclusion result. The proofs are rather technical and are not presented here. However, they are of course part of the formalization.

Proposition 8.5 $f_{dec\_inj\_ren}(\nu\overleftarrow{s_{io\_multi}}) \subseteq \nu\overleftarrow{s_{io\_multi}}$

Proposition 8.6 $((f_{bij\_ren} \circ f_{reach}) \cup id)$ *is* s_{ho_out}*-,* s_{ho_in}*- and* s_{var_multi}*-closed*$^{\top}_{f_{dec\_inj\_ren}}$.

Next, we show the auxiliary lemma based on f_{reach}:

Lemma 8.7 $f_{reach}(\nu\overleftrightarrow{s_{io\_multi}}) \subseteq \nu\overleftrightarrow{s_{io\_multi}}$

Proof Using conditional closedness, we slightly strengthen the claim to make the proof go through. By the closure properties of conditional closedness (Figure 5.1) we get the following derivation.

$$\frac{\begin{array}{ll} f_{dec\_inj\_ren} & \textit{symmetric} \\ ((f_{bij\_ren} \circ f_{reach}) \cup id) & \textit{symmetric} \\ \\ ((f_{bij\_ren} \circ f_{reach}) \cup id) & s_{ho\_out}\text{-}\textit{closed}^{\top}_{f_{dec\_inj\_ren}} \\ ((f_{bij\_ren} \circ f_{reach}) \cup id) & s_{ho\_in}\text{-}\textit{closed}^{\top}_{f_{dec\_inj\_ren}} \\ ((f_{bij\_ren} \circ f_{reach}) \cup id) & s_{var\_multi}\text{-}\textit{closed}^{\top}_{f_{dec\_inj\_ren}} \end{array}}{((f_{bij\_ren} \circ f_{reach}) \cup id)\ s_{io\_multi}\text{-}\textit{closed}^{\top}_{f_{dec\_inj\_ren}}}$$

Symmetry of $f_{dec\_inj\_ren}$ and $((f_{bij\_ren} \circ f_{reach}) \cup id)$ is trivial. Conditional closedness under s_{ho_out}, s_{ho_in} and s_{var_multi} is given by Proposition 8.6. ∎

We now proceed with the proof of the third inclusion.

Lemma 8.8 $\nu\overleftrightarrow{s_{io\_multi}} \subseteq s_{var\_cxt}(\nu\overleftrightarrow{s_{io\_multi}})$

Proof We can now use the fact that f_{reach} behaves oppositely to s_{var_cxt}. Let A and B be two processes with $(A, B) \in \nu\overleftrightarrow{s_{io\_multi}}$. We have to show $(A, B) \in s_{var\_cxt}(\nu\overleftrightarrow{s_{io\_multi}})$. Assume a transition $A \xrightarrow{x} A'$. We have to show the existence of B' s.t. $B \xrightarrow{x} B'$ with $(A', B') \in \nu\overleftrightarrow{s_{io\_multi}}$. By Lemma 8.7 it suffices to show $(A', B') \in f_{reach}(\nu\overleftrightarrow{s_{io\_multi}})$. By Proposition 8.2 (iii) we know that there is an unguarded occurrence of x in A. By definition of s_{io_multi} this means that B has also an unguarded occurrence of x. Again by Proposition 8.2 (iii) we can infer that there is also a transition $B \xrightarrow{x} B'$ for a context B'. We now have $(A', B') \in f_{reach}(\nu\overleftrightarrow{s_{io\_multi}})$ by definition of f_{reach}. ∎

Chapter 9

Formalization in Coq

The entire formalization[1] of the presented results is carried out constructively in the interactive theorem prover Coq without the use of any axioms. In the following we present some of the formalization techniques and tools we have used. See Appendix A for more information about the organization of the Coq files.

Variable-binding

Dealing with de Bruijn indices was much facilitated by Autosubst [15]. The instantiation operation is generated automatically after defining the process terms, with `bind process` indicating where a binder occurs.

```
Inductive process: Type :=
| Send: chan → process → process
| Receive: chan → {bind process} → process
| Var: var → process
| Par: process → process → process
| Nil: process.
```

Autosubst provides a normalization procedure for substitutions. E.g., the substitution $\uparrow \circ (0 :: id)$ will be normalized to id. From the normalization procedure, which can be triggered by the tactic `asimpl`, one obtains a complete decision procedure for equivalence of substitutions. Applying it to a term causes the normalization of every substitution it contains, e.g. the equivalence $\uparrow \circ (0 :: id) = id$ becomes directly solvable after `asimpl` simplifies it to $id = id$. We use this automation extensively, which saves us many manual proofs and makes dealing with substitutions a comfortable task.

Coinductive Reasoning

The definitions of bisimulation and bisimilarity in Coq follow exactly the definitions given in Chapter 3. For a functional s, we define an s-bisimulation as a post-fixed-point of s and bisimilarity νs as the union of all post-fixed-points.

```
Definition postfp (s: relation T → relation T) (x: relation T) :=
  x ⊆ s x.
Definition nu (s: relation T → relation T) (p q: T) :=
```

[1] Available at `https://www.ps.uni-saarland.de/~convent/bachelor.php`

```
∃ x, postfp s x ∧ x p q.
```

The theorem of Knaster-Tarski (Theorem 3.2) assures that νs is the greatest fixed-point. Originally we have worked with the Paco library [4], which defines the greatest fixed-point of a functional through Coq's `CoInductive` construct and enables incremental proofs. Because we have not made use of this incremental feature, which in case of bisimilarity allows to incrementally build up a bisimulation, we opted for the simple definitions as described.

Type Classes

Type classes offer convenience and clearer proofs by automatic inference of arguments when they are needed, under consideration of their context. We profit from this mechanism in that once we have shown certain properties, we do not have to explicitly refer to them anymore but still can rely on them. For example, for a functional we have a proposition certifying its monotonicity. When applying a lemma to this functional, we do not have to refer to its monotonicity explicitly, as it can be determined automatically by type class inference. We use type classes not only for monotonicity, but also for compatibility, conditional closedness and symmetry of a function.

Combining Proof Components

Having proven properties about bisimilarity components, their proofs can be generically merged. In the case of proofs relying on compatibility, applying the closure properties of Figure 4.1 is done automatically through type class inference. For proofs relying on conditional closedness however, merging cannot always be done by type class inference. The reason for this is as follows: To intersect two conditional closedness results (Figure 5.1 (iii)), the two minimum (maximum) conditions need to be lifted (lowered) to a "common denominator" (Figure 5.1 (i)). Because of the two side conditions in Figure 5.1 (i), type class inference does not succeed. However, since in principle this could be automated, we mark this issue as future work.

Multisets

For the definition of s_{var_multi}, we use a multiset implementation which is based on unordered lists of natural numbers. We have oriented ourselves and originally used an implementation provided by the CoLoR library [5] for this. It turned out that for working with multisets we really depend only on lemmas about list manipulation (Coq Standard Library) and two lemmas about finite sums over multisets. Finite sums over multisets are defined through `fold_right`.

```
Definition fin_sum (f: nat → nat) (m: list nat): nat :=
  fold_right (fun x a ⇒ (f x + a)) 0 m.
```

The first lemma reveals our purpose of sums over multisets, namely to characterize the number of occurrences of an unguarded variable x within a substituted process $A[\sigma]$.

$$\mathrm{mul}_x(V(A[\sigma])) = \sum_{y \in V(A)} \mathrm{mul}_x(V(\sigma(y)))$$

The second lemma about finite sums is rather technical but actually the only point where we make use of the fact that the order of a list representing a multiset is irrelevant.

$$V(A) \subseteq V(B) \quad \Rightarrow \quad \sum_{x \in V(A)} f(x) \leq \sum_{x \in V(B)} f(x)$$

Chapter 10

Conclusion

Background

Dealing with variables and substitutions is easy for paper proofs, but can cause much technical work for a formalization in a proof assistant (e.g., in Hirschkoff's Coq formalization of the π-calculus, of the "800 proved lemmas, about 600 are concerned with operators on free names" [3]). The background of this thesis was the idea to investigate how de Bruijn indices, under support of the Autosubst library, can be applied as a formalization technique to the field of concurrency theory. Higher-order process calculi form very good candidates for such an exploration, because de Bruijn indices with Autosubst have already been shown to excel when dealing with higher-order substitutions (e.g. through the formalization of the typed lambda calculus System F [15]).

Final Discussion und Future Work

We have found dealing with variables indeed to be easy, as the support of de Bruijn indices and parallel substitutions through the normalization procedure of Autosubst has taken away much work. In the formalization of Maksimović et al., the authors mention an auxiliary lemma[1] which was "particularly difficult to prove" [7]: It essentially states that "when substituting several variables with empty messages, the substitution order does not matter" [7]. Although we have not shown the result this auxiliary lemma is used for, we observe that through parallel substitutions we do not need to respect the substitution order and its imposed constraints at all.

As described in Section 5.2.2, we have not found a way to extend the notion of compatibility in a way which lets us prove the soundness of up-to techniques for greatest fixed-points abiding by minimum and maximum conditions. At this point, it is not clear to us whether this is possible but we think it might be worthwhile investigating this more.

In our formalization, it turned out to be difficult to show coincidence of our definition of IO bisimilarity through s_{var_cxt} and the definition given by [7] through s_{var_rem}. We suspect that the proof we have given via the bridging definition based on multisets (s_{var_multi}) can be simplified.

[1] Lemma 4 in [7]

For future work, one can extend the shown results by proving decidability of IO bisimilarity, coincidence of IO bisimilarity with other bisimilarities and by giving an axiomatization of IO bisimilarity.

All in all, we hope to contribute to the exploration of formalization techniques in the field of higher-order process calculi, with particular regard to variable-binding and the pursuit of compositional results. We suspect that our framework for giving compositional proofs does not make our whole development smaller. When considering different recombinations of multiple components though (as bisimilarities are given in [6]), we suspect that using compositional results has an impact on the size of the development. Independently of the time and space needed to do the proofs, we see the following main advantage of the compositional approach: The decomposition of monolithic proofs of the main results provides a better understanding, by having the "big picture" at hand but by being able to focus on concrete components at the same time.

Appendix A

Organization of Coq Files

The total development contains 3561 lines of code (not counting external libraries). It was compiled using Coq 8.5 and Autosubst 1.4.

Base/Base.v	Base library from "Introduction to Computational Logic" lecture [16]
Prelim.v	Preliminary definitions
Multiset.v	Definition and properties of multisets
HoCore.v	Definitions of processes and transitions, initialization of Autosubst
ComplLat.v	Definitions and properties of the complete lattice on binary relations
Compat.v	Notion of compatibility, its closure properties and implications
CondClos.v	Notion of conditional closedness, its closure properties and implications
UpToNu.v	Compatibility of the up-to-bisimilarity technique
Ren.v	Properties of renamings and subsets of it
Subst.v	Properties of substitutions
Bis.v	Definitions of bisimilarity components
BisDecInjRen.v	Compatibility for decidable injective renamings
BisSubstit.v	Conditional closedness for substitution closure
BisCongr.v	Conditional closedness for congruence closures
IoCxtBis.v	Results about $\nu\overleftrightarrow{s_{io\_cxt}}$
IoMultiBis.v	Results about $\nu\overleftrightarrow{s_{io\_multi}}$
InclIoCxtBisIoRemBis.v	Inclusion of $\nu\overleftrightarrow{s_{io\_cxt}}$ in $\nu\overleftrightarrow{s_{io\_rem}}$
InclIoRemBisIoMultiBis.v	Inclusion of $\nu\overleftrightarrow{s_{io\_rem}}$ in $\nu\overleftrightarrow{s_{io\_multi}}$
InclIoMultiBisIoCxtBis.v	Inclusion of $\nu\overleftrightarrow{s_{io\_multi}}$ in $\nu\overleftrightarrow{s_{io\_cxt}}$

Bibliography

[1] Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lévy. Explicit substitutions. *J. Funct. Program.*, 1(4):375–416, 1991.

[2] Nicolaas Govert De Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. In *Indagationes Mathematicae (Proceedings)*, volume 75, pages 381–392. Elsevier, 1972.

[3] Daniel Hirschkoff. A full formalisation of pi-calculus theory in the calculus of constructions. In *Proceedings of the 10th International Conference on Theorem Proving in Higher Order Logics*, TPHOLs, pages 153–169. Springer-Verlag, 1997.

[4] Chung-Kil Hur, Georg Neis, Derek Dreyer, and Viktor Vafeiadis. The power of parameterization in coinductive proof. *ACM SIGPLAN Notices*, 48(1):193–206, 2013.

[5] Adam Koprowski. Coq formalization of the higher-order recursive path ordering. *Applicable Algebra in Engineering, Communication and Computing*, 20(5-6):379–425, 2009.

[6] Ivan Lanese, Jorge A. Pérez, Davide Sangiorgi, and Alan Schmitt. On the expressiveness and decidability of higher-order process calculi. *Information and Computation*, 209(2):198–226, 2011.

[7] Petar Maksimović and Alan Schmitt. HOCore in Coq. In *ITP*, volume 9236 of *Lecture Notes in Computer Science*, pages 278–293. Springer, 2015.

[8] Robin Milner. *Communication and concurrency*. PHI Series in computer science. Prentice Hall, 1989.

[9] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I. *Information and Computation*, 100(1):1–40, 1992.

[10] Joachim Parrow, Johannes Borgström, Palle Raabjerg, and Johannes Åman Pohjola. Higher-order psi-calculi. *Mathematical Structures in Computer Science*, 24(2), 2014.

[11] Damien Pous. Complete lattices and up-to techniques. In *APLAS*, volume 4807 of *Lecture Notes in Computer Science*, pages 351–366. Springer, 2007.

[12] Damien Pous. Coinduction all the way up. Preprint, January 2016.

[13] Davide Sangiorgi. *Expressing mobility in process algebras: first-order and higher-order paradigms*. PhD thesis, The University of Edinburgh, 1993.

[14] Davide Sangiorgi. On the bisimulation proof method. *Mathematical Structures in Comp. Sci.*, 8(5):447–479, October 1998.

[15] Steven Schäfer, Tobias Tebbi, and Gert Smolka. Autosubst: Reasoning with de Bruijn terms and parallel substitutions. In *ITP*, volume 9236 of *Lecture Notes in Computer Science*, pages 359–374. Springer, 2015.

[16] Gert Smolka and Chad E. Brown. Introduction to computational logic. Lecture Notes (retrieved from https://www.ps.uni-saarland.de/courses/cl-ss14/script/icl.pdf on August 15, 2016). 2014.

[17] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific J. Math.*, 5(2):285–309, 1955.

[18] Christian Urban. Nominal techniques in Isabelle/HOL. *J. Autom. Reasoning*, 40(4):327–356, 2008.