The Logic of Fault-Tolerance in Message-Passing Concurrency

MSc Thesis *(Afstudeerscriptie)*

written by

Bas van den Heuvel
(born 8 February 1993 in Amsterdam)

under the supervision of **Dr Jorge A. Pérez (RUG)** and **Dr Alban Ponse**, and submitted to the Board of Examiners in partial fulfillment of the requirements for the degree of

MSc in Logic

at the *Universiteit van Amsterdam.*

Date of the public defense:	**Members of the Thesis Committee:**
4 July 2019	Dr Benno van den Berg
	Dr Sung-Shik Jongmans
	Dr Jorge A. Pérez
	Dr Alban Ponse
	Dr Bernardo Toninho
	Prof Dr Yde Venema (chair)

Institute for Logic, Language and Computation

Abstract

Exception handling is a widespread method of making programs fault-tolerant. Exploiting a Curry-Howard correspondence with linear logic, Caires and Pérez have shown how to use modalities denoting non-deterministically available behaviour in the π-calculus to interpret a functional programming language with exception handling. Their work did, however, not address constructs for exception handling in the relevant setting of programming languages with message-passing concurrency. This thesis focusses on this limitation, and uses their ideas to give a new, logically motivated account of exception handling in message-passing concurrency. As key reference in this study, we examine the typed framework for interactional exception handling by Carbone et al., which is fairly expressive, but does not rest upon Curry-Howard foundations. We identify a fragment of this framework on which we can define a conversion to an asynchronous version of the typed π-calculus from Caires and Pérez. Our main result is that source processes and their respective conversions correspond in terms of typing and progress; based on our conversion, we conjecture an operational correspondence result.

Contents

Chapter 1

Introduction

Logic is deeply rooted in many aspects of society, from reasoning and argumentation techniques in philosophy and politics, to scientific research. It plays a particularly prominent role in the field of computer science, for example as basis for decisions in the design of programming languages, or to verify programs against desirable correctness properties, such as termination.

In the twentieth century, great scientists, such as Haskell Curry and William Howard, have discovered that logic may not only serve as a means to reason about programming, but that logic and programming are in fact the same mathematical object. It has been shown that different logics correspond exactly to different classes of programs [14, 15, 31, 16, 34, 13]. An important result of these correspondences is that logical proofs can actually be interpreted as programs, and hence *executed*. This has led to the development of mathematical tools, such as proof assistants. These tools have, for example, enabled mathematicians to prove conjectures that are now theorems, such as the Kepler Conjecture [27, 28] and the Four-Color Theorem [24].

This so-called Curry-Howard correspondence (sometimes called Curry-Howard-De Bruijn correspondence, because De Bruijn independently discovered the correspondence as well) has not only proven to be fruitful from the perspective of logicians. It goes both ways: programs can be interpreted as logical proofs. This means that programmers can use methods from logic to reason about programming. For example, propositional formulas in intuitionistic implicational natural deduction can be used to type terms in the λ-calculus [12], making it possible to verify the consistency of such typed programs. See [41] for a history and overview of the Curry-Howard correspondence.

Another, more recent example of the Curry-Howard correspondence involves Girard's linear logic [22, 23]. This logic was invented to model systems with a fine-grained management of resources. In 2010, Caires and Pfenning showed that this substructural logic can justify a calculus that can be used to model concurrent computation [7]. This calculus is a fragment of the well-known typed π-calculus [35, 38], which is widely accepted as the λ-calculus for interaction and concurrency. The π-calculus models parallel programs that communicate over channels through message passing. These channels are typed with so-called session types that describe the order in which messages will be passed over the channel [29]. The following example is a formula in linear logic that represents the session type of a channel over which a value of type A will be sent ($\otimes$), followed by the reception of a value of type B ($\parr$), after which the channel will be closed ($\mathbf{1}$):

$$A \otimes (B \parr \mathbf{1})$$

The fragment of this calculus that linear logic induces is guaranteed to be free from deadlock: no program will be waiting for another program that is also waiting. The isomorphism between the logic and the calculus exists on two levels: proofs are processes and cut reduction is computation, or, in the case of the π-calculus, communication. Consider, for example, the following cut reduction in linear logic, involving multiplicative conjunction $\otimes$ and its dual multiplicative disjunction $\parr$:

$$\dfrac{\dfrac{\vdash \Delta, A; \Theta \quad \vdash \Delta', B; \Theta}{\vdash \Delta, \Delta', A \otimes B; \Theta}\,(\mathrm{T}\otimes) \quad \dfrac{\vdash \Gamma, \overline{A}, \overline{B}; \Theta}{\vdash \Gamma, \overline{A} \parr \overline{B}; \Theta}\,(\mathrm{T}\parr)}{\vdash \Delta, \Delta', \Gamma; \Theta}\,(\mathsf{Tcut}) \quad \rightarrow \quad \dfrac{\dfrac{\vdash \Delta, A; \Theta \quad \vdash \Gamma, \overline{A}, \overline{B}; \Theta}{\vdash \Delta, \Gamma, \overline{B}; \Theta}\,(\mathsf{Tcut}) \quad \vdash \Delta', B; \Theta}{\vdash \Delta, \Delta', \Gamma; \Theta}\,(\mathsf{Tcut})$$

In the π-calculus, $\overline{x}\langle y\rangle$ denotes the sending of a channel y over channel x, and $x(v)$ the reception of a channel over x which will be used as v in the following process. $P \parallel Q$ denotes parallel composition, and $(\nu x)(P \parallel Q)$ creates a channel x, making one end available in P and the other in Q. If we interpret $\otimes$ as sending, and $\parr$ as receiving, the above cut reduction corresponds to the following computational reduction in the π-calculus:

$$(\nu x)((\nu y)(\overline{x}\langle y\rangle; P \parallel Q) \parallel x(v); R) \rightarrow (\nu x)(P \parallel (\nu y)(Q \parallel R\{y/v\}))$$

The channel y, provided by the process Q, is transferred and becomes available to the process R. After the transfer, the channel v in R has to be renamed to y.

In order to be able to use session type theory to study a wide range of concurrent programming principles, many extensions have been proposed. See [33] for an overview of the development of session types in the past twenty years. After the discovery of the Curry-Howard correspondence with linear logic, such extensions have been re-examined to see to what extent a logical basis can support them. Examples are recursion [39] and polymorphism [20, 6].

Caires and Pérez developed a session type theory based on classical linear logic, extended with types that can be used to indicate non-deterministically available session protocols [4]. The following logical formula represents the newly added monad & to type a channel that might be used to send a value of type A, but it could also be possible that no such behaviour happens at all (i.e. failure):

$$\&(A \otimes \mathbf{1})$$

In order to demonstrate the power of their extension, Caires and Pérez showed how to encode a higher-level functional programming language into their session calculus. This language is similar to the λ-calculus, albeit somewhat artificial. It supports a very limited form of concurrency through the usage of threads (sequences of instructions that can be run independently from the main program), and its main feature is the support of *try-catch blocks*, a well-known construction used to make programs *fault-tolerant*.

$$\mathsf{try}\ P\ \mathsf{catch}\ Q$$

The above line of pseudo-code is executed by running P, in which exceptional behaviour might occur. If it does, an exception is thrown. When that happens, the rest of the try-block gets cancelled, and replaced by Q, the catch-block, also called the *exception handler*. For example, a try-block could contain code to open a user-selected file. If this file does not exist, an exception is thrown, triggering the exception handler, which warns the user to pick a file that does exist.

The logically supported fault-tolerance in functional programming from Caires and Pérez leaves us to wonder what this means for concurrent programming. Can we adopt their ideas to model exception handling in the Curry-Howard fragment of the π-calculus? To what extent can their modality for non-determinism play a role in this? How does this compare to other process calculi with fault-tolerance through exception handling, with or without a logical basis?

In this thesis, we use the types for non-determinism of Caires and Pérez to implement fault-tolerance through exception handling for concurrent programming. We do this by studying the session type system of Carbone et al. [10, 11], which does have fault-tolerance for concurrency, but lacks a logical basis in the form of a Curry-Howard correspondence. The main contributions are

- the identification of a fragment of the system by Carbone et al., suitable for a conversion to a system that does have such a logical foundation;
- the conversion itself; and
- correspondence results between source processes and their conversions.

This offers a first answer to how the session type extension of Caires and Pérez can be used to implement fault-tolerance through exception handling for message-passing concurrency.

We first define the session type calculus we will be using in Chapter 2. This calculus implements the types for non-determinism of Caires and Pérez and a form of asynchronous communication from DeYoung et al. [17]. In Chapter 3, we describe the calculus by Carbone et al., and define a conversion on a fragment of it to our calculus. Chapter 4 contains some suggestions for future work, such as fault-tolerance in multiparty session type systems (used to model message-passing protocols between multiple participants from a global perspective). The thesis is concluded in Chapter 5. In Appendix A we give the typing system from Carbone et al. as in [11], Appendix B contains an example of a converted process, Appendix C contains some diagrams to visualise the idea behind our conversion, and Appendix D gives a multiparty session type calculus extended with a modality for non-deterministic availability.

Chapter 2

An asynchronous session calculus with non-determinism

In this chapter we present our process calculus, which we call DCPTP (for DeYoung, Caires, Pfenning, Toninho, Pérez). It is based on the system by Caires and Pérez [4], which is an extension of the π-calculus [35, 38] with new constructs for non-deterministically available resources. The system of [4] maintains a correspondence to the classical linear logic Σ_2 described by Andreoli [2], based on the work by Girard [22]. By adopting the method by DeYoung et al. [17], the process terms of [4] are changed such that messages are sent asynchronously, allowing for more concurrency. For this, we base our system on the polyadic π-calculus, which allows for communicating multiple values at once. To demonstrate the viability of our process calculus, we show in Section 2.2 that it adequately models a π-calculus using buffers to exhibit asynchronous communication [21, 17, 30, 18] – a concrete and plausibly implementable notion.

Definition 2.1 (Process terms)

Process terms (P, Q) are given by

$$\begin{aligned} P, Q \ ::= \ & (P \parallel Q) \mid (\nu x)P \mid \mathbf{0} \mid [x \leftrightarrow y] \mid x.\overline{\texttt{close}} \mid x.\texttt{close} \\ & \mid \overline{x}\langle y, x'\rangle \mid x(y, x'); P \mid x.\overline{\texttt{inl}}\langle x'\rangle \mid x.\overline{\texttt{inr}}\langle x'\rangle \mid x.\texttt{case}(x')\,(P, Q) \\ & \mid !x(y); P \mid \overline{x}\langle y\rangle \mid x.\overline{\texttt{some}}\langle x'\rangle \mid x.\overline{\texttt{none}} \mid x.\texttt{some}_{\tilde{w}}(x'); P \mid P \oplus Q \end{aligned}$$

where $x, y, \tilde{w}$ are channel names.

In the above definition, $(P \parallel Q)$ denotes parallel composition. It is standard in the π-calculus that if P reduces to P', then $(P \parallel Q)$ reduces to $(P' \parallel Q)$. The term $(\nu x)P$ denotes the creation of a new channel with name x, available to P (i.e. x is bound in P). Actions with an overbar and angled brackets denote sending, e.g. $\overline{x}\langle y, x'\rangle$ ("send y and x' over x") and $x.\overline{\texttt{some}}\langle x'\rangle$ ("send `some` and x' over x"). Actions with round brackets denote receiving, e.g. $x(y, x'); P$ ("receive y and x' over x, continue with P") and $x.\texttt{some}_{\tilde{w}}(x'); P$ ("wait for `some` and x' on x, continue with P"). Besides the ones presented in this section, the standard congruence and reduction rules from [35, 38] apply.

Note that in the π-calculus, it is common that any communication action is a prefix to another process (e.g. $\overline{x}\langle y\rangle; P$). Due to asynchrony, sending actions in our inference rules would never be a prefix to the continuation, so we would always get sending actions such as $\overline{x}\langle y, x'\rangle; \mathbf{0}$. It is common to omit trailing $\mathbf{0}$ processes, so we have decided to present sending actions as singular actions, i.e. not as a prefix.

Definition 2.2 (Types)

Types are given by

$$A, B ::= \mathbf{1} \mid \bot \mid A \otimes B \mid A \parr B \mid A \,\&\, B \mid A \oplus B \mid ?A \mid !A \mid \&A \mid \oplus A$$

A type A represents one end of a channel. The other end of the channel should have a complementary behaviour. For example, if on one end a value is sent, on the other end that value should be received. In our session type system, this is captured by the duality of types: one end of a channel is typed A, while the other end is typed with the dual type $\overline{A}$. Duality is defined with standard DeMorgan-style laws, and corresponds to negation in classical linear logic $A^{\bot}$. It is easy to see that duality is an involution [22]: $\overline{\overline{A}} = A$.

Definition 2.3 (Duality)

The duality relation on types is given by

$\overline{\mathbf{1}} = \bot$	$\overline{A \otimes B} = \overline{A} \parr \overline{B}$	$\overline{A \,\&\, B} = \overline{A} \oplus \overline{B}$	$\overline{?A} = !\overline{A}$	$\overline{\&A} = \oplus\overline{A}$
$\overline{\bot} = \mathbf{1}$	$\overline{A \parr B} = \overline{A} \otimes \overline{B}$	$\overline{A \oplus B} = \overline{A} \,\&\, \overline{B}$	$\overline{!A} = ?\overline{A}$	$\overline{\oplus A} = \&\overline{A}$

2.1 Inference rules and cut reductions as asynchronous communication

Linear logic is commonly presented using a collection of sequent calculus style inference rules. We present our process terms and types by annotating those inference rules with process terms and channel names. A sequent is then of the form $P \vdash \Delta; \Theta$. Here, P is the process term. Δ is the linear context, consisting of pairs $x : A$ where x is a channel name and A its type. Θ is the unrestricted context, consisting of similar channel/type pairs. We sometimes omit the unrestricted context, when it is irrelevant for the rule or explanation. We give a summary of all the rules in Figure 2.1.

By applying these rules, we can obtain process terms with channels of dual types and combine them using (Tcut). This makes it possible to apply linear logic cut reduction. The resulting transformations of process terms in the sequents correspond to the computational semantics of our process calculus. Therefore, in order to demonstrate the Curry-Howard correspondence, our process reductions are presented along with their respective cut reductions. We summarize all process reductions in Figure 2.2.

We end the section with two example processes (see "Example 1: division by zero" and "Example 2: exception handling").

Note that many rules use Greek capital letters to represent arbitrary collections of channel/type pairs. In order to be consistent with cut reduction examples, we generally use Δ for sending rules, and Γ for receiving ones. If an inference rule has different Greek letters throughout its premises and conclusion, they are meant to represent disjoint collections of channel/type pairs, e.g. $\Delta \cap \Gamma = \emptyset$. If the same letter appears throughout premises and conclusion, it is meant to represent the same collection of channel/type pairs in each occurrence. In essence, these collections place requirements on the linear context for inferencing. Also, some rules have requirements on channel names appearing in collections. This is to ensure that a channel is or is not free in a process. We often omit these requirements after presenting them the first time.

Weakening and contraction. Because unrestricted resources need not be consumed, the unrestricted context, which is used for persistent processes, can be filled at will using the *weakening rule* (T<). Redundant unrestricted resources can be merged using the *contraction rule* (T>). It is, however, not possible to completely remove a resource from this context once it has been added. Note that in session type literature, it is common to use these rules implicitly.

$$\frac{P \vdash \Delta; \Theta}{P \vdash \Delta; x : A, \Theta}\ (\mathrm{T}<) \qquad \frac{P \vdash \Delta; x : A, y : A, \Theta}{P \vdash \Delta; x : A, \Theta}\ (\mathrm{T}>)$$

Composition. There are two ways of composing processes. If the processes do not have channels in common, they can be composed to run in parallel using the *mix rule* (T$\|$). However, if processes have channels of dual types, they can be composed with the *cut rule* (Tcut). This binds the common channel, removing it from the linear context.

$$\frac{P \vdash \Delta; \Theta \quad Q \vdash \Gamma; \Theta}{P \parallel Q \vdash \Delta, \Gamma; \Theta}\ (\mathrm{T}\|) \qquad \frac{P \vdash \Delta, x : \overline{A}; \Theta \quad Q \vdash \Gamma, x : A; \Theta}{(\nu x)\,(P \parallel Q) \vdash \Delta, \Gamma; \Theta}\ (\mathsf{Tcut})$$

Because duality is an involution ($\overline{\overline{A}} = A$), the rule (Tcut) is symmetric. This corresponds to the commutativity of parallel composition in the π-calculus (which also holds for our calculus): $P \parallel Q$ and $Q \parallel P$ denote the same unique process (formally, $P \parallel Q \equiv Q \parallel P$). The strength of the correspondence between our calculus and linear logic is reflected in the fact that symmetric uses of (Tcut) are congruent in linear logic:

$$\frac{\vdash \Delta, A; \Theta \quad \vdash \Gamma, \overline{A}; \Theta}{\vdash \Delta, \Gamma; \Theta}\ (\mathsf{Tcut}) \ \cong\ \frac{\vdash \Gamma, \overline{A}; \Theta \quad \vdash \Delta, A; \Theta}{\vdash \Gamma, \Delta; \Theta}\ (\mathsf{Tcut})$$

The empty process and forwarding. The empty process $\mathbf{0}$ does not have any behaviour, so it can only be introduced along with an empty linear context. This can be done using the *empty axiom* (T$\cdot$). The forwarding process $[x \leftrightarrow y]$ mimics the behaviour of channel x on channel y. The types of x and y are dual: everything that goes into x comes out of y and vice versa. The *identity axiom* (Tid) introduces this term. Again, due to the involutivity of duality, forwarding is symmetric: $[x \leftrightarrow y] \equiv [y \leftrightarrow x]$. Note that these rules allow an arbitrary unrestricted context.

$$\frac{}{\mathbf{0} \vdash \cdot; \Theta}\ (\mathrm{T}\cdot) \qquad \frac{}{[x \leftrightarrow y] \vdash x : A, y : \overline{A}; \Theta}\ (\mathsf{Tid})$$

Since the empty process does not have any behaviour, there are no process reductions associated with it. However, the process equivalence $P \parallel \mathbf{0} \equiv P$ does correspond to the following proof congruence:

$$\dfrac{\dfrac{}{\vdash \cdot; \Theta}\,(\mathrm{T}\cdot) \quad \vdash \Delta; \Theta}{\vdash \Delta; \Theta}\,(\mathrm{T}\parallel) \;\cong\; \vdash \Delta; \Theta$$

Our first process reduction involves (Tid). A process P that acts on a channel x that is being forwarded to a channel y can be changed to directly act on y by renaming all occurrences of x in P to y:

$$(\nu x)\,(P \parallel [x \leftrightarrow y]) \to P\{y/x\}$$

This corresponds to the following cut reduction:

$$\dfrac{\vdash \overline{A}; \Theta \quad \dfrac{}{\vdash A, \overline{A}; \Theta}\,(\mathsf{Tid})}{\vdash \overline{A}; \Theta}\,(\mathsf{Tcut}) \;\to\; \vdash \overline{A}; \Theta$$

Closing channels. Types $\mathbf{1}$ and its dual $\bot$ represent channels that can be closed. The *termination axiom* (T$\mathbf{1}$) closes the channel on one end, and allows an arbitrary unrestricted context. After closing a channel, the process terminates, so no continuation is allowed. On the other end of a channel, it is necessary to wait for it to close. This can be done using the *closing rule* (T$\bot$). In standard π-calculus, the waiting action is a prefix to the remaining process (i.e. the conclusion of (T$\bot$) is $x.\mathtt{close}; P$). However, since the closing channel cannot be used in the remaining process, we can increase concurrency by waiting for the channel to close in parallel with the continuation.

$$\dfrac{}{x.\overline{\mathtt{close}} \vdash x : \mathbf{1}; \Theta}\,(\mathrm{T}\mathbf{1}) \qquad \dfrac{P \vdash \Delta; \Theta \quad x \notin \Delta}{x.\mathtt{close} \parallel P \vdash x : \bot, \Delta; \Theta}\,(\mathrm{T}\bot)$$

If a process is waiting for a channel to close, while a parallel process closes it, the channel can actually be closed, leaving only the continuation process:

$$(\nu x)\,\big(x.\overline{\mathtt{close}} \parallel (x.\mathtt{close} \parallel P)\big) \to P$$

This corresponds to the following cut reduction in linear logic:

$$\dfrac{\dfrac{}{\vdash \mathbf{1}; \Theta}\,(\mathrm{T}\mathbf{1}) \quad \dfrac{\vdash \Delta; \Theta}{\vdash \bot, \Delta; \Theta}\,(\mathrm{T}\bot)}{\vdash \Delta; \Theta}\,(\mathsf{Tcut}) \;\to\; \vdash \Delta; \Theta$$

Sending and receiving. If a channel has type $A \otimes B$, another channel of type A will be sent over it, after which it continues to behave as type B. To optimize concurrency in the involved process, we want to send the A channel asynchronously. Our first attempt at the *send rule* (T$\otimes'$), also used as a first attempt in [17], is as follows:

$$\frac{R \vdash \Delta, y : A; \Theta \quad Q \vdash \Delta', x : B; \Theta \quad x \notin \Delta, y \notin \Delta'}{(\nu y)\,(\overline{x}\langle y\rangle \parallel R \parallel Q) \vdash \Delta, \Delta', x : A \otimes B; \Theta}\,(\mathrm{T}\otimes')$$

This rule is, however, flawed, as explained in [17]. For example, if the continuation Q sends on x, that action will be put in parallel with the first send action. Now if this process would be composed with a process receiving on x, two possible reductions could take place: the communication of the first send action, or that of the second. This violates the guarantee that messages are sent in order. Another problematic situation would be if Q were to receive on x. Reduction would then allow the just sent message to be wrongly received by Q.

We implement the solution to this from [17], which is to continue as B on a new channel, which is sent along with the channel of type A. This send rule (T$\otimes$) guarantees that any consecutive communication actions happen on another channel, maintaining the order of messages.

$$\frac{R \vdash \Delta, y : A; \Theta \quad Q \vdash \Delta', x' : B; \Theta \quad x, x' \notin \Delta, y \notin \Delta'}{(\nu y)(\nu x')\,(\overline{x}\langle y, x'\rangle \parallel R \parallel Q) \vdash \Delta, \Delta', x : A \otimes B; \Theta}\,(\mathrm{T}\otimes)$$

The dual of $A \otimes B$ is $\overline{A} \parr \overline{B}$. A channel of this type will receive a channel of type $\overline{A}$ and continue as $\overline{B}$. The *receive rule* (T$\parr$) receives the channel and the continuation channel from (T$\otimes$) simultaneously.

$$\frac{P \vdash \Gamma, u : \overline{A}, x' : \overline{B}; \Theta \quad x \notin \Gamma}{x(u, x'); P \vdash \Gamma, x : \overline{A} \parr \overline{B}; \Theta}\,(\mathrm{T}\parr)$$

When two parallel processes simultaneously send and receive a channel y over the same channel, y is sent and becomes available to the receiving process P. Since the rule (T$\otimes$) dictates that y cannot be used in Q after sending it, y only needs to be shared between P and the process providing y, R. Both remaining processes continue their communications on the now shared continuation channel.

$$(\nu x)\,((\nu y)(\nu x')\,(\overline{x}\langle y, x'\rangle \parallel R \parallel Q) \parallel x(u, x'); P) \to (\nu x')\,((\nu y)\,(R \parallel P\{y/u\}) \parallel Q)$$

This corresponds to the following cut reduction:

$$\dfrac{\dfrac{\vdash \Delta, A; \Theta \quad \vdash \Delta', B; \Theta}{\vdash \Delta, \Delta', A \otimes B; \Theta}\,(\mathrm{T}\otimes) \quad \dfrac{\vdash \Gamma, \overline{A}, \overline{B}; \Theta}{\vdash \Gamma, \overline{A} \parr \overline{B}; \Theta}\,(\mathrm{T}\parr)}{\vdash \Delta, \Delta', \Gamma; \Theta}\,(\mathsf{Tcut}) \quad \to \quad \dfrac{\dfrac{\vdash \Delta, A; \Theta \quad \vdash \Gamma, \overline{A}, \overline{B}; \Theta}{\vdash \Delta, \Gamma, \overline{B}; \Theta}\,(\mathsf{Tcut}) \quad \vdash \Delta', B; \Theta}{\vdash \Delta, \Delta', \Gamma; \Theta}\,(\mathsf{Tcut})$$

Branching. A channel typed $A \,\&\, B$ offers a choice between two behaviours A and B. A channel of dual type $\overline{A} \oplus \overline{B}$ chooses which branch to take. Since the channel type after the choice is already known at the choosing side beforehand, it is possible to add concurrency by sending the selection asynchronously. However, similar problems to those with asynchronous sending might occur. Therefore, we again send a continuation channel along with the choice. These types can be inferred with the *offer rule* (T&) and the *left and right select rules* (T$\oplus_1$) and (T$\oplus_2$).

$$\frac{P_1 \vdash \Gamma, x' : A; \Theta \quad P_2 \vdash \Gamma, x' : B; \Theta \quad x \notin \Gamma}{x.\mathtt{case}(x')(P_1, P_2) \vdash \Gamma, x : A \,\&\, B; \Theta}\ (\mathrm{T}\&)$$

$$\frac{Q \vdash \Delta, x' : \overline{A}; \Theta \quad x \notin \Delta}{(\nu x')\left(x.\overline{\mathtt{inl}}\langle x'\rangle \parallel Q\right) \vdash \Delta, x : \overline{A} \oplus \overline{B}; \Theta}\ (\mathrm{T}\oplus_1)$$

$$\frac{Q \vdash \Delta, x' : \overline{B}; \Theta \quad x \notin \Delta}{(\nu x')\left(x.\overline{\mathtt{inr}}\langle x'\rangle \parallel Q\right) \vdash \Delta, x : \overline{A} \oplus \overline{B}; \Theta}\ (\mathrm{T}\oplus_2)$$

When a choice is offered while a choice is made on the same channel in parallel, a process reduces to the chosen branch composed with the choosing side's continuation process. Both processes then continue their communications on the now shared continuation channel.

$$(\nu x)\left((\nu x')\left(x.\overline{\mathtt{inl}}\langle x'\rangle \parallel Q\right) \parallel x.\mathtt{case}(x')\,(P_1, P_2)\right) \to (\nu x')\,(Q \parallel P_1)$$
$$(\nu x)\left((\nu x')\left(x.\overline{\mathtt{inr}}\langle x'\rangle \parallel Q\right) \parallel x.\mathtt{case}(x')\,(P_1, P_2)\right) \to (\nu x')\,(Q \parallel P_2)$$

The left choice process reduction corresponds to the following cut reduction (the right choice reduction is analogous):

$$\dfrac{\dfrac{\dfrac{\vdash \Delta, \overline{A}; \Theta}{\vdash \Delta, \overline{A} \oplus \overline{B}; \Theta}\ (\mathrm{T}\oplus_1) \quad \dfrac{\vdash \Gamma, A; \Theta \quad \vdash \Gamma, B; \Theta}{\vdash \Gamma, A \,\&\, B; \Theta}\ (\mathrm{T}\&)}{\vdash \Delta, \Gamma; \Theta}\ (T\mathsf{cut})}{} \quad \to \quad \dfrac{\vdash \Delta, \overline{A}; \Theta \quad \vdash \Gamma, A; \Theta}{\vdash \Delta, \Gamma; \Theta}\ (\mathsf{Tcut})$$

By consecutively applying rules (T&), (T$\oplus_1$) and (T$\oplus_2$), it is possible to model a choice between more than two branches. This is a common practice in session type calculi. We can formally generalise this notion by defining n-ary choice:

$$\frac{P_i \vdash \Gamma, x' : A_i; \Theta \qquad (\text{for all } i \in I) \quad x \notin \Gamma}{x.\mathtt{case}(x')\,(\mathtt{l}_i : P_i)_{i \in I} \vdash \Gamma, x : \&\{\mathtt{l}_i : A_i\}_{i \in I}; \Theta}\ (\mathrm{T}\&_I)$$

$$\frac{Q \vdash \Delta, x' : \overline{A_i}; \Theta \quad x \notin \Delta}{(\nu x')\left(x.\overline{\mathtt{l}_i}\langle x'\rangle \parallel Q\right) \vdash \Delta, x : \oplus\{\mathtt{l}_i : \overline{A_i}\}_{i \in I}; \Theta}\ (\mathrm{T}\oplus_{i \in I})$$

The process reduction then looks as follows:

$$(\nu x)\left((\nu x')\left(x.\overline{\mathtt{l}_i}\langle x'\rangle \parallel Q\right) \parallel x.\mathtt{case}(x')\,(\mathtt{l}_i : P_i)_{i \in I}\right) \to (\nu x')\,(Q \parallel P_i)$$

Persistence. To be able to model a situation in which a server concurrently offers a service to any amount of clients, one can use *persistent processes*. This is where the unrestricted context comes in: in order to continuously offer the same service, the resources needed for it must not run out.

A channel typed $!A$ offers a persistent service of type A. By using the *persistent offer rule* (T!), a persistent process receives a channel on which the service will take place. The dual channel type $?\overline{A}$ merely exists to make a persistent service available to clients using (Tcut). This type can be introduced using the *persistent request rule* (T?), which indeed does not have any effect on the process term.

$$\frac{P \vdash u : A; \Theta}{!x(u); P \vdash x : !A; \Theta}\,(\mathrm{T}!) \qquad \frac{Q \vdash \Delta; x : \overline{A}, \Theta}{Q \vdash \Delta, x : ?\overline{A}; \Theta}\,(\mathrm{T}?)$$

To request a service that is persistently offered, one can use the *copy rule* (Tcopy). This rule lets the client send a channel on which the service will take place. Sending this channel asynchronously does not pose the same problems as before. The channel x over which the client sends its channel y is persistent and can only be used to request services. The service happens over the sent channel y. This service channel y can be interpreted as a continuation channel, thus saving us from problems in the order of messages. The *persistent cut rule* (T$\mathsf{cut}^?$) composes a persistent service with a possible client.

$$\frac{Q \vdash \Delta, y : \overline{A}; x : \overline{A}, \Theta}{(\nu y)\,(\overline{x}\langle y\rangle \parallel Q) \vdash \Delta; x : \overline{A}, \Theta}\,(\mathrm{T}\mathsf{copy})$$

$$\frac{Q \vdash \Delta; x : \overline{A}, \Theta \quad P \vdash u : A; \Theta}{(\nu x)\,(Q \parallel !x(u); P) \vdash \Delta; \Theta}\,(\mathrm{T}\mathsf{cut}^?)$$

Offering and requesting a service in parallel spawns the service and copies the service offer for further requests:

$$(\nu x)\,((\nu y)\,(\overline{x}\langle y\rangle \parallel Q) \parallel !x(u); P) \to (\nu x)\,((\nu y)\,(P\{y/u\} \parallel Q) \parallel !x(u); P)$$

This corresponds to the following cut reduction:

$$\dfrac{\dfrac{\dfrac{\vdash \Delta, \overline{A}; \overline{A}, \Theta}{\vdash \Delta; \overline{A}, \Theta}\,(\mathrm{T}\mathsf{copy})}{\vdash \Delta, ?\overline{A}; \Theta}\,(\mathrm{T}?) \quad \dfrac{\vdash A; \Theta}{\vdash !A; \Theta}\,(\mathrm{T}!)}{\vdash \Delta; \Theta}\,(\mathrm{T}\mathsf{cut}) \qquad \to \qquad \dfrac{\dfrac{\vdash A; \Theta}{\vdash A; \overline{A}, \Theta}\,(\mathrm{T}<) \quad \vdash \Delta, \overline{A}; \overline{A}, \Theta}{\vdash \Delta; \overline{A}, \Theta}\,(\mathrm{T}\mathsf{cut}) \quad \vdash A; \Theta}{\vdash \Delta; \Theta}\,(\mathrm{T}\mathsf{cut}^?)$$

Non-determinism. The following rules are asynchronous versions of those from [4]. When operations fail or resources turn out to be unavailable (e.g. network connection is lost or memory is full), a graceful method of cancellation is desired. The type $\&A$ models possible failure. By sending a message over a channel x of that type, one can indicate whether the behaviour A is available. The *some rule* ($\mathrm{T}\&_d^x$) announces availability and continues to provide the promised behaviour. The *none rule* ($\mathrm{T}\&^x$) announces unavailability, so it disallows any further behaviour. We add concurrency by sending the announcement asynchronously, so, as before, it is necessary to send a continuation channel along with the message in order to prevent message order violations.

$$\frac{P \vdash \Delta, x' : A; \Theta \quad x \notin \Delta}{(\nu x')\,(x.\overline{\texttt{some}}\langle x'\rangle \parallel P) \vdash \Delta, x : \&A; \Theta}\ (\mathrm{T}\&_d^x)$$

$$\frac{}{x.\overline{\texttt{none}} \vdash x : \&A; \Theta}\ (\mathrm{T}\&^x)$$

These rules can be used to model explicitly waiting for a resource to become available (or not). One could argue that this is a reason not to send the availability announcement asynchronously: the resource might be used on the sending side before the announcement is even received. However, when constructing process terms using our inference rules, availability is known beforehand. Moreover, communications considering the available resource can only happen after the availability announcement has been received and the receiving process is ready to communicate over the continuation channel.

The channel on the receiving side has dual type $\oplus\overline{A}$. The *non-deterministic select rule* ($\mathrm{T}\oplus_{\tilde{w}}^x$) waits for the complementary process to announce availability. It requires all channels in the linear context, $\tilde{w}$, to be of type $\&-$, such that in case the resource is not available, the process can be gracefully shut down by announcing non-availability over all those channels.

$$\frac{Q \vdash \tilde{w} : \&\Gamma, x' : \overline{A}; \Theta \quad x \notin \tilde{w}}{x.\texttt{some}_{\tilde{w}}(x'); Q \vdash \tilde{w} : \&\Gamma, x : \oplus\overline{A}; \Theta}\ (\mathrm{T}\oplus_{\tilde{w}}^x)$$

The process reductions associated with these types depend on what rule the ($\mathrm{T}\oplus_{\tilde{w}}^x$) rule is composed with. If it is composed with rule ($\mathrm{T}\&_d^x$), the expected behaviour is available, so both processes continue their computations:

$$(\nu x)\,((\nu x')\,(x.\overline{\texttt{some}}\langle x'\rangle \parallel P) \parallel x.\texttt{some}_{\tilde{w}}(x'); Q) \to (\nu x')\,(P \parallel Q)$$

The corresponding cut reduction is:

$$\frac{\dfrac{\vdash \Delta, A; \Theta}{\vdash \Delta, \&A; \Theta}\,(\mathrm{T}\&_d^x) \quad \dfrac{\vdash \&\Gamma, \overline{A}; \Theta}{\vdash \&\Gamma, \oplus\overline{A}; \Theta}\,(\mathrm{T}\oplus_{\tilde{w}}^x)}{\vdash \Delta, \&\Gamma; \Theta}\,(\mathsf{Tcut}) \qquad \to \qquad \frac{\vdash \Delta, A; \Theta \quad \vdash \&\Gamma, \overline{A}; \Theta}{\vdash \Delta, \&\Gamma; \Theta}\,(\mathsf{Tcut})$$

If it is composed with rule (T&^{x}), the expected behaviour is not available, so both processes terminate, and all remaining channels in the linear context of the receiving process are concurrently announced to be unavailable.

$$(\nu x)\,(x.\overline{\texttt{none}} \parallel x.\texttt{some}_{\tilde{w}}(x'); Q) \to w_1.\overline{\texttt{none}} \parallel \cdots \parallel w_n.\overline{\texttt{none}}$$

This corresponds to the following cut reduction:

$$\dfrac{\dfrac{}{\vdash \&A; \Theta}\,(\mathrm{T}\&^x) \quad \dfrac{\vdash \&\Gamma, \overline{A}; \Theta}{\vdash \&\Gamma, \oplus\overline{A}; \Theta}\,(\mathrm{T}\oplus^x_{\tilde{w}})}{\vdash \&\Gamma; \Theta}\,(\mathsf{Tcut}) \qquad \to \qquad \dfrac{\dfrac{}{\vdash \&A; \Theta}\,(\mathrm{T}\&^{w_1}) \quad \cdots \quad \dfrac{}{\vdash \&A; \Theta}\,(\mathrm{T}\&^{w_n})}{\vdash \&\Gamma; \Theta}\,(\mathrm{T}\|)$$

As explained before, when constructing process terms, one needs to decide whether the resource is actually available to apply above rules. This is only non-determinism in the sense that the receiving side and reductions can take care of both availability and unavailability. It is, however, possible to determine the availability by looking at the complementary sending process. This limits the possibilities of modelling true non-determinism.

To achieve true non-determinism, we define the *non-deterministic composition rule* (T&$_{nd}$), also from [4]. It combines two processes with identical contexts.

$$\dfrac{P \vdash \&\Delta; \Theta \quad Q \vdash \&\Delta; \Theta}{P \oplus Q \vdash \&\Delta; \Theta}\,(\mathrm{T}\&_{nd})$$

Reductions happen *within* the composition, so if $P \to P'$, then $P \oplus Q \to P' \oplus Q$, and similarly for Q. Non-deterministic composition is distributive, due to the following process equivalence:

$$(\nu x)\,((P \oplus Q) \parallel R) \equiv (\nu x)\,(P \parallel R) \oplus (\nu x)\,(Q \parallel R)$$

This corresponds to the following proof congruence:

$$\dfrac{\dfrac{\vdash \&\Delta, \&A; \Theta \quad \vdash \&\Delta, \&A; \Theta}{\vdash \&\Delta, \&A; \Theta}\,(\mathrm{T}\&_{nd}) \quad \vdash \&\Gamma, \oplus\overline{A}; \Theta}{\vdash \&\Delta, \&\Gamma; \Theta}\,(\mathsf{Tcut})$$

$$\cong$$

$$\dfrac{\dfrac{\vdash \&\Delta, \&A; \Theta \quad \vdash \&\Gamma, \oplus\overline{A}; \Theta}{\vdash \&\Delta, \&\Gamma; \Theta}\,(\mathsf{Tcut}) \quad \dfrac{\vdash \&\Delta, \&A; \Theta \quad \vdash \&\Gamma, \oplus\overline{A}; \Theta}{\vdash \&\Delta, \&\Gamma; \Theta}\,(\mathsf{Tcut})}{\vdash \&\Delta, \&\Gamma; \Theta}\,(\mathrm{T}\&_{nd})$$

By combining above availability rules and the non-deterministic composition, one can model true non-deterministic availability. Consider, for example, processes P and Q such that $P \vdash x' : A$ and $Q \vdash x' : \overline{A}, \tilde{w} : \&\Delta$. We make x non-deterministically available by i) applying (T&^x_d) to P, (T$\oplus^x_{\tilde{w}}$) to Q, and composing them using (Tcut); ii) applying (T&^{x}) to P, (T$\oplus^x_{\tilde{w}}$) to Q, and composing them using (Tcut); and non-deterministically composing the results from i) and ii) using (T&$_{nd}$). The result of this composition reduces as follows:

$$(\nu x)\,((\nu x')\,(x.\overline{\texttt{some}}\langle x'\rangle \parallel P) \parallel x.\texttt{some}_{\tilde{w}}; Q) \oplus (\nu x)\,(x.\overline{\texttt{none}} \parallel x.\texttt{some}_{\tilde{w}}; Q)$$
$$\to (\nu x')\,(P \parallel Q) \oplus (w_1.\overline{\texttt{none}} \parallel \cdots \parallel w_n.\overline{\texttt{none}})$$

Notice how there are actually two reductions going on here: one within the left and the other within the right of the non-deterministic composition.

Example 1: division by zero

Suppose we have a process D that receives a natural number and sends the result of dividing 5 by that number: $D \vdash y : \mathsf{Div} = \mathsf{nat} \parr (\mathsf{nat} \otimes \bot)$

It is unclear what would happen if we send the number 0. In order to gracefully handle such a situation, we make a safe version of D. It uses a process Z which receives a natural number, and returns it if it is not equal to 0. Otherwise, it cancels the communication with `none`:
$Z \vdash z : \mathsf{NZero} = \mathsf{nat} \parr (\&(\mathsf{nat} \otimes \bot))$

Our safe process SafeD will use D and Z as services, on respective channels y' and z'. The goal is to receive a natural number, and if it is not equal to 0, to send the result of 5 divided by the number. Otherwise, the `none` from Z should cascade to our process, gracefully cancelling it and the process expecting a result from it.

$$\mathsf{SafeD} \vdash x : \mathsf{nat} \parr (\&(\mathsf{nat} \otimes \bot); y' : \overline{\mathsf{Div}}, z' : \overline{\mathsf{NZero}}$$

For clarity, we annotate the lines of the process definition with their meaning.

$\mathsf{SafeD} =$	
$x(u, x_1); (\nu z)(\overline{z'}\langle z\rangle \parallel$	receive value, connect to Z
$\quad (\nu u')(\nu z_1)([u \leftrightarrow u'] \parallel \overline{z}\langle u', z_1\rangle \parallel$	send value to Z
$\quad\quad z_1.\mathtt{some}_{(x_1)}(z_2); z_2(v, z_3); (z_3.\overline{\mathtt{close}} \parallel \mathsf{SafeD}_1)$	(possibly) receive value from Z
$\quad)$	and close connection
$)$	
$\mathsf{SafeD}_1 =$	
$(\nu y)(\overline{y'}\langle y\rangle \parallel$	connect to D
$\quad (\nu v')(\nu y_1)([v \leftrightarrow v'] \parallel \overline{y}\langle v', y_1\rangle \parallel$	send value to D
$\quad\quad y_1(w, y_2); (y_2.\overline{\mathtt{close}} \parallel \mathsf{SafeD}_2)$	receive division result from D
$\quad)$	and close connection
$)$	
$\mathsf{SafeD}_2 =$	
$(\nu x_2)(x_1.\overline{\mathtt{some}}\langle x_2\rangle \parallel$	announce successful computation
$\quad (\nu w')(\nu x_3)([w \leftrightarrow w'] \parallel \overline{x_2}\langle w', x_3\rangle \parallel x_3.\mathtt{close})$	send division result
$)$	close connection

If a process $P \vdash x : \mathsf{nat} \otimes (\oplus(\mathsf{nat} \parr \mathbf{1}))$ sends a non-zero number, the following process (we omit the proper composition with services D and Z) reduces successfully: $(\nu x)(P \parallel \mathsf{SafeD}) \to^* P'$, where P' is some unrelated continuation of P. However, if P sends 0, the division does not happen at all:

$$(\nu x)(P \parallel \mathsf{SafeD}) \to^* (\nu x_1)(x_1.\mathtt{some}_{\tilde{w}} \parallel x_1.\overline{\mathtt{none}}) \to \mathbf{0}$$

Here $\to^*$ denotes the transitive, reflexive closure of $\to$.

Example 2: exception handling

This example showcases the exception handling as Caires and Pérez implement it for their functional programming language [4]. The idea is that an exception "router" waits for an expression to indicate success or failure. In a successful situation the router receives the result of the expression's computation and passes it to the continuation of the program. In case of a failure, the router invokes the exception handler instead, which computes a result of the same type as expected in a successful situation. Again, the router passes on this result to the continuation.

The fault-tolerant programming language of Caires and Pérez allows an exception to be thrown, accompanied with a value. This value can be used in the exception handler. However, to keep the example readable, we give a version of the exception handling principle without accompanying such a value. The source program looks like $\texttt{TRY}\, e_1\, \texttt{CATCH}\, e_2$, where e_1 is either $\texttt{LIFT}\, e_1'$ (in case of success), or `THROW` (in case of failure). We given an asynchronous version of the encoding of Caires and Pérez, where y is the channel on which a continuation can expect the result of the try-catch block.

$$
\begin{array}{ll}
[\![\texttt{TRY}\, e_1\, \texttt{CATCH}\, e_2]\!]_y = & \\
(\nu j)(& \text{the exception router} \\
\quad (\nu k)(& \text{(possibly) will pass value to context} \\
\quad\quad [\![e_1]\!]_{k,j} \parallel & \text{the expression} \\
\quad\quad k.\texttt{some}_{\emptyset}(k_1); k_1(v, k_2); & \text{(possibly) receive value} \\
\quad\quad k_2(z, k_3); (k_3.\texttt{close} \parallel & \text{receive router and close} \\
\quad\quad\quad (\nu z_1)(z.\overline{\texttt{inl}}\langle z_1\rangle \parallel & \text{signal success} \\
\quad\quad\quad\quad (\nu v')(\nu z_2)([v \leftrightarrow v'] \parallel \overline{z_1}\langle v', z_2\rangle \parallel z_2.\overline{\texttt{close}}) & \text{send value to router} \\
\quad\quad\quad) & \\
\quad\quad) & \\
\quad) \parallel & \\
\quad j.\texttt{case}(j_1)(& \\
\quad\quad j_1(v, j_2); (j_2.\overline{\texttt{close}} \parallel & \text{receive value} \\
\quad\quad\quad (\nu v')(\nu y_1)([v \leftrightarrow v'] \parallel \overline{y}\langle v', y_1\rangle \parallel y_1.\overline{\texttt{close}}) & \text{send value to continuation} \\
\quad\quad), & \\
\quad\quad j_1.\texttt{close} \parallel [\![e_2]\!]_y & \text{invoke exception handler} \\
\quad) & \\
) &
\end{array}
$$

The two possible encodings of e_1 are given on the next page.

$[\![\texttt{LIFT}\, e_1']\!]_{k,j} =$

$(\nu q)($	will receive value from expression
$\quad [\![e_1']\!]_q \parallel$	the expression
$\quad q(v, q_1); (q_1.\texttt{close} \parallel$	receive value
$\quad\quad (\nu k_1)(k.\overline{\texttt{some}}\langle k_1\rangle \parallel$	signal success
$\quad\quad\quad (\nu v')k_2([v \leftrightarrow v'] \parallel \overline{k_1}\langle v', k_2\rangle \parallel$	send value
$\quad\quad\quad\quad (\nu j')(\nu k_3)([j \leftrightarrow j'] \parallel \overline{k_2}\langle j', k_3\rangle \parallel k_3.\overline{\texttt{close}})$	pass on router
$\quad\quad\quad)$	
$\quad\quad)$	
$\quad)$	
$)$	

$[\![\texttt{THROW}]\!]_{k,j} =$

$k.\overline{\texttt{none}} \parallel (\nu j_1)(j.\overline{\texttt{inr}}\langle j_1\rangle \parallel j_1.\overline{\texttt{close}})$	cancel value passing and signal failure

$$\frac{P \vdash \Delta;\Theta}{P \vdash \Delta; x:A,\Theta}\ (\mathrm{T}<) \qquad \frac{P \vdash \Delta; x:A, y:A,\Theta}{P \vdash \Delta; x:A,\Theta}\ (\mathrm{T}>)$$

$$\frac{P \vdash \Delta;\Theta \quad Q \vdash \Gamma;\Theta}{P \parallel Q \vdash \Delta,\Gamma;\Theta}\ (\mathrm{T}\|) \qquad \frac{P \vdash \Delta, x:\overline{A};\Theta \quad Q \vdash \Gamma, x:A;\Theta}{(\nu x)\,(P \parallel Q) \vdash \Delta,\Gamma;\Theta}\ (\mathsf{Tcut})$$

$$\frac{}{\mathbf{0} \vdash \cdot;\Theta}\ (\mathrm{T}\cdot) \qquad \frac{}{[x \leftrightarrow y] \vdash x:A, y:\overline{A};\Theta}\ (\mathsf{Tid})$$

$$\frac{}{x.\overline{\mathtt{close}} \vdash x:\mathbf{1};\Theta}\ (\mathrm{T}\mathbf{1}) \qquad \frac{P \vdash \Delta;\Theta \quad x \notin \Delta}{x.\mathtt{close} \parallel P \vdash x:\bot,\Delta;\Theta}\ (\mathrm{T}\bot)$$

$$\frac{R \vdash \Delta, y:A;\Theta \quad Q \vdash \Delta', x':B;\Theta \quad x,x' \notin \Delta, y \notin \Delta'}{(\nu y)(\nu x')\,(\overline{x}\langle y,x'\rangle \parallel R \parallel Q) \vdash \Delta,\Delta', x:A \otimes B;\Theta}\ (\mathrm{T}\otimes)$$

$$\frac{P \vdash \Gamma, u:\overline{A}, x':\overline{B};\Theta \quad x \notin \Gamma}{x(u,x'); P \vdash \Gamma, x:\overline{A} \parr \overline{B};\Theta}\ (\mathrm{T}\parr)$$

$$\frac{P_1 \vdash \Gamma, x':A;\Theta \quad P_2 \vdash \Gamma, x':B;\Theta \quad x \notin \Gamma}{x.\mathtt{case}(x')(P_1,P_2) \vdash \Gamma, x:A \,\&\, B;\Theta}\ (\mathrm{T}\&)$$

$$\frac{Q \vdash \Delta, x':\overline{A};\Theta \quad x \notin \Delta}{(\nu x')\,(x.\overline{\mathtt{inl}}\langle x'\rangle \parallel Q) \vdash \Delta, x:\overline{A} \oplus \overline{B};\Theta}\ (\mathrm{T}\oplus_1)$$

$$\frac{Q \vdash \Delta, x':\overline{B};\Theta \quad x \notin \Delta}{(\nu x')\,(x.\overline{\mathtt{inr}}\langle x'\rangle \parallel Q) \vdash \Delta, x:\overline{A} \oplus \overline{B};\Theta}\ (\mathrm{T}\oplus_2)$$

$$\frac{P_i \vdash \Gamma, x':A_i;\Theta \qquad (\text{for all } i \in I) \quad x \notin \Gamma}{x.\mathtt{case}(x')\,(\mathtt{l}_i : P_i)_{i\in I} \vdash \Gamma, x:\&\{\mathtt{l}_i : A_i\}_{i\in I};\Theta}\ (\mathrm{T}\&_I)$$

$$\frac{Q \vdash \Delta, x':\overline{A_i};\Theta \quad x \notin \Delta}{(\nu x')\,(x.\overline{\mathtt{l}_i}\langle x'\rangle \parallel Q) \vdash \Delta, x:\oplus\{\mathtt{l}_i : \overline{A_i}\}_{i\in I};\Theta}\ (\mathrm{T}\oplus_{i\in I})$$

$$\frac{P \vdash u:A;\Theta}{!x(u); P \vdash x:!A;\Theta}\ (\mathrm{T}!) \qquad \frac{Q \vdash \Delta; x:\overline{A},\Theta}{Q \vdash \Delta, x:?\overline{A};\Theta}\ (\mathrm{T}?)$$

$$\frac{Q \vdash \Delta, y:\overline{A}; x:\overline{A},\Theta}{(\nu y)\,(\overline{x}\langle y\rangle \parallel Q) \vdash \Delta; x:\overline{A},\Theta}\ (\mathsf{Tcopy}) \qquad \frac{Q \vdash \Delta; x:\overline{A},\Theta \quad P \vdash u:A;\Theta}{(\nu x)\,(Q \parallel !x(u); P) \vdash \Delta;\Theta}\ (\mathsf{Tcut}^?)$$

$$\frac{P \vdash \Delta, x':A;\Theta \quad x \notin \Delta}{(\nu x')\,(x.\overline{\mathtt{some}}\langle x'\rangle \parallel P) \vdash \Delta, x:\&A;\Theta}\ (\mathrm{T}\&^x_d) \qquad \frac{}{x.\overline{\mathtt{none}} \vdash x:\&A;\Theta}\ (\mathrm{T}\&^x)$$

$$\frac{Q \vdash \tilde{w}:\&\Gamma, x':\overline{A};\Theta \quad x \notin \tilde{w}}{x.\mathtt{some}_{\tilde{w}}(x'); Q \vdash \tilde{w}:\&\Gamma, x:\oplus\overline{A};\Theta}\ (\mathrm{T}\oplus^x_{\tilde{w}})$$

$$\frac{P \vdash \&\Delta;\Theta \quad Q \vdash \&\Delta;\Theta}{P \oplus Q \vdash \&\Delta;\Theta}\ (\mathrm{T}\&_{nd})$$

Figure 2.1: Typed process inference rules.

$$
\begin{aligned}
(\nu x)\,(P \parallel [x \leftrightarrow y]) &\to P\{y/x\} \\
(\nu x)\,\big(x.\overline{\texttt{close}} \parallel x.\texttt{close} \parallel P\big) &\to P \\
(\nu x)\,((\nu y)(\nu x')\,(\overline{x}\langle y, x'\rangle \parallel R \parallel Q) \parallel x(u, x'); P) &\to (\nu x')\,((\nu y)\,(R \parallel P\{y/u\}) \parallel Q) \\
(\nu x)\,\big((\nu x')\,\big(x.\overline{\texttt{inl}}\langle x'\rangle \parallel Q\big) \parallel x.\texttt{case}(x')\,(P_1, P_2)\big) &\to (\nu x')\,(Q \parallel P_1) \\
(\nu x)\,\big((\nu x')\,\big(x.\overline{\texttt{inr}}\langle x'\rangle \parallel Q\big) \parallel x.\texttt{case}(x')\,(P_1, P_2)\big) &\to (\nu x')\,(Q \parallel P_2) \\
(\nu x)\,\big((\nu x')\,\big(x.\overline{\texttt{l}_i}\langle x'\rangle \parallel Q\big) \parallel x.\texttt{case}(x')\,(\texttt{l}_i : P_i)_{i\in I}\big) &\to (\nu x')\,(Q \parallel P_i) \\
(\nu x)\,((\nu y)\,(\overline{x}\langle y\rangle \parallel Q) \parallel\, !x(u); P) &\to (\nu x)\,((\nu y)\,(P\{y/u\} \parallel Q) \parallel\, !x(u); P) \\
(\nu x)\,((\nu x')\,(x.\overline{\texttt{some}}\langle x'\rangle \parallel P) \parallel x.\texttt{some}_{\tilde{w}}(x'); Q) &\to (\nu x')\,(P \parallel Q) \\
(\nu x)\,(x.\overline{\texttt{none}} \parallel x.\texttt{some}_{\tilde{w}}(x'); Q) &\to w_1.\overline{\texttt{none}} \parallel \cdots \parallel w_n.\overline{\texttt{none}} \\
\text{if } P \to P', \text{ then } P \oplus Q &\to P' \oplus Q \\
\text{if } Q \to Q', \text{ then } P \oplus Q &\to P \oplus Q'
\end{aligned}
$$

Figure 2.2: Process reductions.

2.2 Asynchrony through buffers

A more common way of exhibiting asynchronous communication is through the use of buffers [21, 17, 30, 18]. Following the work of [17], in order to demonstrate that the use of floating messages is adequate, we want to show that it is capable of modelling buffered communication. Therefore, we define a more common version of the π-calculus in which sending actions are synchronous, but the semantics (i.e. process reductions) are defined using buffers. Then, we show how this calculus can be modelled by our asynchronous calculus by giving a relation between the two calculi and showing how the reductions are perfectly matched.

The process terms are given in Definition 2.4, which also contains the definition of buffers. The usual rules for structural congruence apply. Buffers are directed, so the buffer $z[\overline{m}\rangle x$ receives messages on z and outputs on x. Empty buffers do not commit to a direction, so $z[\rangle x \equiv z\langle]x$. Reductions are given in Definition 2.5. They are separated into a sending and a receiving group. These semantics show that, although sending actions are blocking, messages can always immediately be placed into an available buffer. Note that, for a clear presentation, reductions are given for untyped terms. However, they can only be applied to well-typed terms (i.e. all involved channel names have to be bound and processes need to exhibit complementary behaviour on opposite sides of channels).

Definition 2.4 (Process terms with buffers)

Process terms (P, Q) with buffers are given by

$$\begin{aligned} m &::= \mathtt{close} \mid y \mid \mathtt{inl} \mid \mathtt{inr} \mid !y \mid \mathtt{some} \mid \mathtt{none} \\ P, Q &::= (P \parallel Q) \mid (\nu x)P \mid \mathbf{0} \mid x.\overline{\mathtt{close}} \mid x.\mathtt{close}; P \\ &\quad \mid \overline{x}\langle y\rangle; P \mid x(y); P \mid x.\overline{\mathtt{inl}}; P \mid x.\overline{\mathtt{inr}}; P \mid x.\mathtt{case}(P, Q) \\ &\quad \mid \overline{x}\langle !y\rangle; P \mid !x(y); P \mid x.\overline{\mathtt{some}}; P \mid x.\overline{\mathtt{none}} \mid x.\mathtt{some}_{\tilde{w}}; P \mid P \oplus Q \\ &\quad \mid x[\overline{m}\rangle y \end{aligned}$$

where $x, y, z, \tilde{w}$ are channel names.

Definition 2.5 (Buffered reductions)

The (untyped) reductions of the buffered calculus are given by

$$z.\overline{\mathtt{close}} \parallel z[\overline{m}\rangle x \to_{\mathsf{b}} z[\mathtt{close}, \overline{m}\rangle x \qquad \text{(S-close)}$$

$$\overline{z}\langle y\rangle; Q \parallel z[\overline{m}\rangle x \to_{\mathsf{b}} Q \parallel z[y, \overline{m}\rangle x \qquad \text{(S-chan)}$$

$$z.\overline{\mathtt{inl}}; Q \parallel z[\overline{m}\rangle x \to_{\mathsf{b}} Q \parallel z[\mathtt{inl}, \overline{m}\rangle x \qquad \text{(S-inl)}$$

$$z.\overline{\mathtt{inr}}; Q \parallel z[\overline{m}\rangle x \to_{\mathsf{b}} Q \parallel z[\mathtt{inr}, \overline{m}\rangle x \qquad \text{(S-inr)}$$

$$\overline{z}\langle\, !y\rangle; Q \parallel z[\overline{m}\rangle x \to_{\mathsf{b}} Q \parallel z[\,!y, \overline{m}\rangle x \qquad \text{(S-pers)}$$

$$z.\overline{\mathtt{some}}; Q \parallel z[\overline{m}\rangle x \to_{\mathsf{b}} Q \parallel z[\mathtt{some}, \overline{m}\rangle x \qquad \text{(S-some)}$$

$$z.\overline{\mathtt{none}} \parallel z[\overline{m}\rangle x \to_{\mathsf{b}} z[\mathtt{none}, \overline{m}\rangle x \qquad \text{(S-none)}$$

$$z[\mathtt{close}\rangle x \parallel x.\mathtt{close}; P \to_{\mathsf{b}} P \qquad \text{(R-close)}$$

$$z[\overline{m}, y\rangle x \parallel x(u); P \to_{\mathsf{b}} z[\overline{m}\rangle x \parallel P\{y/u\} \qquad \text{(R-chan)}$$

$$z[\overline{m}, \mathtt{inl}\rangle x \parallel x.\mathtt{case}(P_1, P_2) \to_{\mathsf{b}} z[\overline{m}\rangle x \parallel P_1 \qquad \text{(R-inl)}$$

$$z[\overline{m}, \mathtt{inr}\rangle x \parallel x.\mathtt{case}(P_1, P_2) \to_{\mathsf{b}} z[\overline{m}\rangle x \parallel P_2 \qquad \text{(R-inr)}$$

$$z[\overline{m}, \,!y\rangle x \parallel !x(u); P \to_{\mathsf{b}} z[\overline{m}\rangle x \parallel !x(u); P \parallel P\{y/u\} \qquad \text{(R-pers)}$$

$$z[\overline{m}, \mathtt{some}\rangle x \parallel x.\mathtt{some}_{\tilde{w}}; P \to_{\mathsf{b}} z[\overline{m}\rangle x \parallel P \qquad \text{(R-some)}$$

$$z[\mathtt{none}\rangle x \parallel x.\mathtt{some}_{\tilde{w}}; P \to_{\mathsf{b}} w_1.\overline{\mathtt{none}} \parallel \cdots \parallel w_n.\overline{\mathtt{none}} \qquad \text{(R-none)}$$

$$\text{if } P \to_{\mathsf{b}} P', \text{ then } P \oplus Q \to_{\mathsf{b}} P' \oplus Q \qquad (\mathrm{ND}_1)$$

$$\text{if } Q \to_{\mathsf{b}} Q', \text{ then } P \oplus Q \to_{\mathsf{b}} P \oplus Q' \qquad (\mathrm{ND}_2)$$

$$\text{if } P \to_{\mathsf{b}} P', \text{ then } P \parallel Q \to_{\mathsf{b}} P' \parallel Q \qquad (\mathrm{Par}_1)$$

$$\text{if } Q \to_{\mathsf{b}} Q', \text{ then } P \parallel Q \to_{\mathsf{b}} P \parallel Q' \qquad (\mathrm{Par}_2)$$

$$\text{if } P \to_{\mathsf{b}} P', \text{ then } (\nu x)P \to_{\mathsf{b}} (\nu x)P' \qquad \text{(Res)}$$

The relation between these buffered process terms and our asynchronous ones is given in Definition 2.6 on page 23 (we give it after the proof of Theorem 2.1, because we motivate some of the assignments there). A message in a buffer can be seen as a floating send action. Therefore, the left and right terms of a sending reduction using buffers are both related to equivalent terms in the asynchronous calculus. Consequently, reductions in which a buffered message is received are related to the synchronization of a floating send and a receive action.

The following theorem is based on [17, Thm. 2, p. 240].

Theorem 2.1

Take a well-typed buffered process term P from Definition 2.4.

i) If $P \to_{\mathsf{b}} Q$ through an R-rule (receiving), then there are processes P' and Q' such that $P =\!\!=^* P' \to Q' \equiv Q$.
ii) If $P \to_{\mathsf{b}} Q$ through an S-rule (sending), then there are process P' and Q' such that $P =\!\!=^* P' \equiv Q' =\!\!=^* Q$.

We use $\to_{\mathsf{b}}$ as in Definition 2.5, $\to$ as in Section 2.1, and $=\!\!=$ as in Definition 2.6. $=\!\!=^*$ denotes the transitive, reflexive closure of $=\!\!=$.

Proof. Because it is the most representative example, we start with the rule for receiving a channel:

$$z[\overline{m}, y\rangle x \parallel x(u); P \to_{\mathsf{b}} z[\overline{m}\rangle x \parallel P\{y/u\} \tag{2.1}$$

Well-typed, the left term looks like this:

$$(\nu x)\left((\nu z)\left(Q \parallel (\nu y)\left(R \parallel z[\overline{m}, y\rangle x\right)\right) \parallel x(u); P\right) \tag{2.2}$$

We need to relate the buffered message to a floating send:

$$(\nu z)(Q \parallel (\nu y)(R \parallel z[\overline{m}, y\rangle x)) =\!\!= (\nu y)(\nu x')(\overline{x}\langle y, x'\rangle \parallel R \parallel (\nu z)(Q \parallel z[\overline{m}\rangle x'))$$

Also, we need to relate the receiving action and the reception of a continuation channel:

$$x(u); P =\!\!= x(u, x'); P\{x'/x\}$$

This relates our well-typed term (2.2) to the following term, using a floating send instead of a buffered messages:

$$(\nu x)\left((\nu y)(\nu x')\left(\overline{x}\langle y, x'\rangle \parallel R \parallel (\nu z)\left(Q \parallel z[\overline{m}\rangle x')\right) \parallel x(u, x'); P\{x'/x\}\right)$$

This matches the left term of the send/receive reduction in the asynchronous system (modulo α-conversion), so we can apply it:

$$\to (\nu x')(\nu y)\left(R \parallel (\nu z)\left(Q \parallel z[\overline{m}\rangle x'\right) \parallel P\{y/u, x'/x\}\right)$$

This term is structurally congruent to

$$(\nu x)(\nu y)\left(R \parallel (\nu z)\left(Q \parallel z[\overline{m}\rangle x\right) \parallel P\{y/u\}\right),$$

which is the right term $z[\overline{m}\rangle x \parallel P\{y/u\}$ of (2.1), but well-typed.

This shows that the reduction in (2.1) can be modelled exactly by relating to our asynchronous calculus and applying the respective reduction from the previous section. The rest of the receiving reduction rules require similar relations and the proofs of their correspondence to our asynchronous system are analogous. This proves statement (i). We are only left with showing our result for the sending reductions.

The rule we present is the reduction for sending a channel, which is a sufficiently representative example for this proof:

$$\overline{z}\langle y\rangle; Q \parallel z[\overline{m}\rangle x \rightarrow_{\mathsf{b}} Q \parallel z[y, \overline{m}\rangle x \tag{2.3}$$

By defining a few additional process term relations, we can show that the left and right terms are equivalent in our asynchronous system. The first new relation is for processes composed with an empty buffer:

$$(\nu z)(P \parallel z[\rangle x) \rightleftharpoons P\{x/z\}$$

The other relation is for sending a continuation channel along with the channel:

$$\overline{x}\langle y\rangle; P\big) \rightleftharpoons (\nu x')(\overline{x}\langle y, x'\rangle \parallel P\{x'/x\})$$

We show by induction on the amount of messages in the buffer that the left and right terms of (2.3) are structurally congruent. For the base case, we make the left term of (2.3) well-typed, and assume its buffer is empty. Then, we apply our relation:

$$\begin{aligned}
&(\nu x)\left((\nu z)\left((\nu y)\left(R \parallel \overline{z}\langle y\rangle; Q\right) \parallel z[\rangle x\right) \parallel P\right) \\
&\rightleftharpoons (\nu x)\left((\nu y)\left(R \parallel \overline{x}\langle y\rangle; Q\{x/z\}\right) \parallel P\right) \\
&\rightleftharpoons (\nu x)\left((\nu y)\left(R \parallel (\nu x')\left(\overline{x}\langle y, x'\rangle \parallel Q\{x'/z\}\right)\right) \parallel P\right) \\
&\rightleftharpoons (\nu x)\left((\nu y)\left(R \parallel (\nu x')\left(\overline{x}\langle y, x'\rangle \parallel (\nu z)\left(Q \parallel z[\rangle x')\right)\right)\right) \parallel P\right) \\
&\equiv (\nu x)\left((\nu y)(\nu x')\left(\overline{x}\langle y, x'\rangle \parallel R \parallel (\nu z)\left(Q \parallel z[\rangle x')\right)\right) \parallel P\right) \\
&\rightleftharpoons (\nu x)\left((\nu z)\left(Q \parallel (\nu y)\left(R \parallel z[y\rangle x\right)\right) \parallel P\right)
\end{aligned}$$

Indeed, this the well-typed form of the right term of (2.3).

For the induction step, we have an induction hypothesis:

$$(\nu x)\left((\nu z)\left((\nu y)\left(R \parallel \overline{z}\langle y\rangle; Q\right) \parallel z[\overline{m}\rangle x\right) \parallel P\right) \overset{\text{IH}}{\equiv} (\nu x)\left((\nu z)\left(Q \parallel (\nu y)\left(R \parallel z[y, \overline{m}\rangle x\right)\right) \parallel P\right)$$

This step is shown by cases on the type of message appended to the buffer. However, since these cases can be proved similarly, we only show the channel name case. We use the induction hypothesis and our relation to connect the left term and the right term of (2.3), well-typed and with an extra channel name in the buffer.

$$\begin{aligned}
&(\nu x)\left((\nu z)\left((\nu y)\left(R \parallel \overline{z}\langle y\rangle; Q\right) \parallel (\nu w)\left(W \parallel z[\overline{m}, w\rangle x\right)\right) \parallel P\right) \\
&\rightleftharpoons (\nu x)\left((\nu w)(\nu x')\left(\overline{x}\langle w, x'\rangle \parallel W \parallel (\nu z)\left((\nu y)\left(R \parallel \overline{z}\langle y\rangle; Q\right) \parallel z[\overline{m}\rangle x'\right)\right) \parallel P\right) \\
&\overset{\text{IH}}{\equiv} (\nu x)\left((\nu w)(\nu x')\left(\overline{x}\langle w, x'\rangle \parallel W \parallel (\nu z)\left(Q \parallel (\nu y)\left(R \parallel z[y, \overline{m}\rangle x'\right)\right)\right) \parallel P\right) \\
&\rightleftharpoons (\nu x)\left((\nu z)\left(Q \parallel (\nu w)\left(W \parallel (\nu y)\left(R \parallel z[y, \overline{m}, w\rangle x\right)\right)\right) \parallel P\right)
\end{aligned}$$

This proves statement (ii). □

Definition 2.6 (Asynchronous process relations)

Let $\circeq$ be the smallest symmetric relation on typed processes defined as:

$$
\begin{aligned}
(\nu z)\,(P \parallel z[\rangle x) &\circeq P\{x/z\} \\
(\nu z)\,(z[\mathtt{close}\rangle x) &\circeq x.\overline{\mathtt{close}} \\
x.\mathtt{close}; P &\circeq x.\mathtt{close} \parallel P \\
(\nu z)\,(Q \parallel (\nu y)\,(R \parallel z[\overline{m}, y\rangle x)) &\circeq (\nu y)(\nu x')\,(\overline{x}\langle y, x'\rangle \parallel R \parallel (\nu z)\,(Q \parallel z[\overline{m}\rangle x')) \\
\overline{x}\langle y\rangle; P &\circeq (\nu x')\,(\overline{x}\langle y, x'\rangle \parallel P\{x'/x\}) \\
x(u); P &\circeq x(u, x'); P\{x'/x\} \\
(\nu z)\,(Q \parallel z[\overline{m}, \mathtt{inl}\rangle x) &\circeq (\nu x')\,\big(x.\overline{\mathtt{inl}}\langle x'\rangle \parallel (\nu z)\,(Q \parallel z[\overline{m}\rangle x')\big) \\
(\nu z)\,(Q \parallel z[\overline{m}, \mathtt{inr}\rangle x) &\circeq (\nu x')\,\big(x.\overline{\mathtt{inr}}\langle x'\rangle \parallel (\nu z)\,(Q \parallel z[\overline{m}\rangle x')\big) \\
x.\overline{\mathtt{inl}}; P &\circeq (\nu x')\,\big(x.\overline{\mathtt{inl}}\langle x'\rangle \parallel P\{x'/x\}\big) \\
x.\overline{\mathtt{inr}}; P &\circeq (\nu x')\,\big(x.\overline{\mathtt{inr}}\langle x'\rangle \parallel P\{x'/x\}\big) \\
x.\mathtt{case}(P_1, P_2) &\circeq x.\mathtt{case}(x')\,(P_1\{x'/x\}, P_2\{x'/x\}) \\
(\nu z)\,(Q \parallel (\nu y)\,(R \parallel z[\overline{m}, !y\rangle x)) &\circeq (\nu y)\,(\overline{x}\langle y\rangle \parallel R \parallel (\nu z)\,(Q \parallel z[\overline{m}\rangle x)) \\
\overline{x}\langle !y\rangle; P &\circeq \overline{x}\langle y\rangle \parallel P \\
(\nu z)\,(Q \parallel z[\overline{m}, \mathtt{some}\rangle x) &\circeq (\nu x')\,(x.\overline{\mathtt{some}}\langle x'\rangle \parallel (\nu z)\,(Q \parallel z[\overline{m}\rangle x')) \\
(\nu z)\,(z[\mathtt{none}\rangle x) &\circeq x.\overline{\mathtt{none}} \\
x.\overline{\mathtt{some}}; P &\circeq (\nu x')\,(x.\overline{\mathtt{some}}\langle x'\rangle \parallel P\{x'/x\}) \\
x.\mathtt{some}_{\tilde{w}}; P &\circeq x.\mathtt{some}_{\tilde{w}}(x'); P\{x'/x\}
\end{aligned}
$$

Chapter 3

Fault-tolerance through exception handling

In this chapter we investigate how the non-determinism modalities introduced in the previous chapter can be applied for fault-tolerance. Carbone et al. introduced a binary session calculus in which one can design processes that can throw exceptions which will be handled in all the appropriate places [10, 11]. After introducing this system, we define a conversion from a fragment of it to the type system we defined in Chapter 2. We finish the chapter in Section 3.4 by proving and conjecturing results about the relation between this fragment and its conversion.

3.1 The fault-tolerant calculus CYH

Processes in the system by Carbone et al. (we call this CYH for Carbone, Yoshida, Honda) consist of services and requests, inside of which exceptions can be thrown. They are accompanied by exception handler processes, which are invoked when an exception is thrown.

CYH has a type system with judgements which we represent in a slightly different form than in [10, 11]. We prefer our simplified representation, because it is sufficient to reason about our conversion. We include a copy of the original type system from [11] in Appendix A.

As a convention, we write $\mathcal{D}$ and $\mathcal{G}$ to represent order-preserving lists of channel names. Judgments are of the following form:

$$\mathcal{G} \vdash P^{\mathcal{D}} \triangleright (x : \alpha_x)_{x \in \mathcal{D}}$$

Here $\mathcal{G}$ is the list of service channels over which requests can be done. $P^{\mathcal{D}}$ is our notation for a process in which every channel $x \in \mathcal{D}$ is free. The judgement says that every channel $x \in \mathcal{D}$ has behaviour α_x. A behavioural type α can be plain (unprotected) or protected. Protected types are of the form $\alpha\{\!|\beta|\!\}$, where α is the default behaviour, and β is the behaviour after an exception is thrown. We often omit the list of service channels and only display the type of a single channel.

A service looks as follows:

$$*a(s)[P^{(s)}, Q^{(s)}]$$

Here $P^{(s)}$ is the default handler, and $Q^{(s)}$ is the exception handler. When a service is requested over channel a, the process P^s is replicated and connected to the requesting process. Exceptions inside both the service and the request replace the replicated default handler with the exception handler.

A request looks as follows, where $s \in \mathcal{D}$:

$$\overline{a}(s)\left[\mathcal{D}, P^{\mathcal{D}}, Q^{\mathcal{D}}\right]$$

Again, $P^{\mathcal{D}}$ is the default handler, and $Q^{\mathcal{D}}$ is the exception handler. When an exception is thrown on either side of any of the channels in $\mathcal{D}$, the exception handler is invoked, as well as those of the processes on the other sides of the channels in $\mathcal{D}$. The type system of CYH has the following rule about default and exception handlers (for both services and requests), which says that the behaviour of a channel after an exception has been thrown should be captured in the type of the channel in the default handler:

$$P^{\mathcal{D}} \triangleright x : \alpha\{[\beta]\} \implies Q^{\mathcal{D}} \triangleright x : \beta$$

The process of requesting a service inside an already established service or request is called *refinement*. It replaces the current exception handler by a new exception handler that handles the newly connected channel as well as those channels that were already connected. The following example shows a refinement of the exception handler of the established request over a, after the number 5 has been sent ($x_1!\langle 5\rangle$). The exception handler R of the request over b refines the original exception handler Q:

$$\overline{a}(x_1)\left[(x_1), x_1!\langle 5\rangle.\overline{b}(x_2)[(x_1, x_2), P^{(x_1,x_2)}, R^{(x_1,x_2)}], Q^{(x_1)}\right]$$

The exception handlers have the following requirement: the behaviour of a channel after an exception has to be the same after refinement, i.e.:

$$\forall x \in \mathcal{D}. Q^{\mathcal{D}} \triangleright x : \beta \implies R^{\mathcal{D}\cup(x)} \triangleright x : \beta$$

Here, Q is the original exception handler, and R is the one that refines it.

When an exception is thrown (with the **throw** process), it affects parallel processes. Their scope, however, is restricted by enclosing request or service definitions. So,

$$\overline{a}(s)[\mathcal{D}, P^{\mathcal{D}}, Q^{\mathcal{D}}] \parallel \textbf{throw} \rightharpoonup Q^{\mathcal{D}},$$

but

$$\overline{a}(s)[\mathcal{D}, P^{\mathcal{D}}, Q^{\mathcal{D}}] \parallel \overline{b}(s')[\mathcal{D}', \textbf{throw}, R^{\mathcal{D}'}] \rightharpoonup \overline{a}(s)[\mathcal{D}, P^{\mathcal{D}}, Q^{\mathcal{D}}] \parallel R^{\mathcal{D}'}.$$

Note the use of $\rightharpoonup$ instead of the usual $\rightarrow$ for reductions. This is because the semantics of CYH is defined with buffers for asynchronous communication, and meta-reductions and runtime processes for reductions. The reductions here are merely to demonstrate the outside behaviour of CYH.

3.2 The fragment CYH$^{\mathsf{CH}}$

The purpose of our intension to define a conversion from CYH processes to DCPTP processes is to demonstrate the power of `some` and `none`. Exceptions can be thrown at any point in a process and they affect multiple parts of the process. The general idea of our conversion is therefore to synchronize all the parts that might affect each other, at each step of communication. We will use `some` to indicate the absence of an exception, whereas we will use `none` to cancel processes when an exception is thrown. Have a look at Appendix C for a visualization of this process.

We do, however, need the ability to invoke exception handlers. This means that we cannot simply cancel all processes with `none`. This issue reflects the non-compositionality of CYH. Consider, for example, the following two processes that are compositionally the same, but in which the exception has different effects:

1. $\overline{a}(x_1)\left[(x_1), \overline{b}(x_2)[(x_2), \textbf{throw} \parallel P^{(x_2)}, Q^{(x_2)}] \parallel R^{(x_1)}, S^{(x_1)}\right]$
   $\rightharpoonup \overline{a}(x_1)\left[(x_1), R^{(x_1)}, S^{(x_1)}\right] \parallel Q^{(x_2)}$
2. $\overline{a}(x_1)\left[(x_1), \textbf{throw} \parallel \overline{b}(x_2)[(x_2), P^{(x_2)}, Q^{(x_2)}] \parallel R^{(x_1)}, S^{(x_1)}\right]$
   $\rightharpoonup Q^{(x_2)} \parallel S^{(x_1)}$

This places a limit on the extent to which we can convert CYH processes. Therefore, we define a fragment of CYH on which we define our conversion, and call it CYH$^{\mathsf{CH}}$ (for Curry-Howard). The definition is given in Def. 3.1 on the next page. It consists of a series of service definitions followed by a series of requests. The session inside a service cannot make requests, but can throw exceptions. Exception handlers can neither make requests nor throw exceptions (this is a restriction already present in CYH). Sessions inside requests or refinements can throw exceptions, and do allow making requests, up to the point where multiple channels are used in parallel. We motivate our choices after the definition.

Limitation 1: Parallelism

A notable limitation of our fragment is that refinement is not allowed inside parallel composition. This is, however, a limitation of CYH, since it is a syntactical assumption under the name *consistent refinement*. This assumption states that whenever a refinement is applied to an already connected channel, the refinement should include *all* channels that are already connected.

The consistent refinement assumption prevents ambiguous situations such as in the following example:

$$\overline{a}(x_2)\left[(x_1, x_2), P^{(x_1)} \parallel \overline{b}(x_3)[(x_2, x_3), \textbf{throw}, Q^{(x_2,x_3)}], R^{(x_1,x_2)}\right]$$
$$\rightharpoonup \overline{a}(x_2)\left[(x_1, x_2), P^{(x_1)} \parallel Q^{(x_2,x_3)}, R^{(x_1,x_2)}\right]$$

After the exception has been thrown, the channel x_3 is not protected anymore, even though it is inside a request. Since x_2 is subject to exceptions, so is x_3.

Definition 3.1

The fragment $\mathsf{CYH}^{\mathsf{CH}}$ is defined by the following, annotated grammar.

$$\mathsf{CYH}^{\mathsf{CH}} ::= \mathsf{SERVICES}^{\emptyset}$$

$$\mathsf{SERVICES}^{\mathcal{G}} ::= (\nu a)\left(*a(s)\left[\mathsf{SESSION}^{\emptyset,(s)}_{\checkmark}, \mathsf{SESSION}^{\emptyset,(s)}_{\times}\right] \parallel \mathsf{SERVICES}^{\mathcal{G}\cup(a)}\right) \mid \mathsf{REQUESTS}^{\mathcal{G}}$$

$$\mathsf{REQUESTS}^{\mathcal{G}} ::= \overline{a}(s)\left[(s), \mathsf{SESSION}^{\mathcal{G},(s)}_{\checkmark}, \mathsf{SESSION}^{\emptyset,(s)}_{\times}\right] \mid \mathsf{REQUESTS}^{\mathcal{G}} \parallel \mathsf{REQUESTS}^{\mathcal{G}} \quad \text{where } a \in \mathcal{G}$$

$$\mathsf{SESSION}^{\mathcal{G},\mathcal{D}}_{\tau} ::= \mathbf{0} \mid$$

$$\left.\begin{array}{l} x?(y).\mathsf{SESSION}^{\mathcal{G},\mathcal{D}}_{\tau} \mid x!\langle e\rangle.\mathsf{SESSION}^{\mathcal{G},\mathcal{D}}_{\tau} \mid \\ x \triangleright \{l_i : \mathsf{SESSION}^{\mathcal{G},\mathcal{D}}_{\tau}\}_{i\in I} \mid x \triangleleft l.\mathsf{SESSION}^{\mathcal{G},\mathcal{D}}_{\tau} \mid \end{array}\right\} \quad \text{where } x \in \mathcal{D}$$

$$\mathsf{SESSION}^{\emptyset,\mathcal{D}_1}_{\tau} \parallel \mathsf{SESSION}^{\emptyset,\mathcal{D}_2}_{\tau} \mid \quad \text{where } \mathcal{D}_1 \cap \mathcal{D}_2 = \emptyset \text{ and } \mathcal{D}_1 \cup \mathcal{D}_2 = \mathcal{D}$$

$$\overline{a}(s)\left[\mathcal{D}\cup(s), \mathsf{SESSION}^{\mathcal{G},\mathcal{D}\cup(s)}_{\checkmark}, \mathsf{SESSION}^{\emptyset,\mathcal{D}\cup(s)}_{\times}\right] \mid \quad \text{where } s \notin \mathcal{D} \text{ and } a \in \mathcal{G}$$

$$\mathbf{throw} \quad \text{where } \tau = \checkmark$$

Here, $\mathcal{D}$ is an order-preserving list of session channel names, $\mathcal{G}$ is a list of service channel names, and $\tau \in \{\checkmark, \times\}$ indicates the possibility of throwing exceptions. The prefix $x?(y)$ is input of y over x, $x!\langle e\rangle$ is output of expression e over x, $\triangleright$ offers a choice, and $\triangleleft$ selects one.

Consistent refinement does not prohibit what we call *disjoint refinement*, in which a request can be used to connect to a new channel without refining the currently connected channels. The following example is typable in CYH. Once the exception is thrown, the exception handler is not susceptible to exceptions anymore.

$$\begin{aligned} &\overline{a}(x_1)\left[(x_1), P^{(x_1)} \parallel \overline{b}(x_2)[(x_2), \mathbf{throw}, Q^{(x_2)}], R^{(x_1)}\right] \\ &\rightharpoonup \overline{a}(x_1)[(x_1), P^{(x_1)}, R^{(x_1)}] \parallel Q^{(x_2)} \end{aligned}$$

As exceptions have an effect across parallel composition, our conversion will need to synchronize those parallel processes. This is where we run into limitations of using `none`. Consider the following process, which contains a disjoint refinement:

$$\overline{a}\left[(x_1), x_1!\langle 5\rangle.\mathbf{throw} \parallel \overline{b}[(x_2), x_2?(y).x_2 \triangleleft l_3.\mathbf{0}, R^{(x_2)}], Q^{(x_1)}\right] \tag{3.1}$$

On both sides of the parallel composition, no exceptions are thrown, so in our conversion the two parallel processes synchronize and the output on x_1 and input on x_2 occur. Indeed, the semantics of CYH allows the process to reduce like this:

$$\rightharpoonup \overline{a}\left[(x_1), \mathbf{throw} \parallel \overline{b}[(x_2), x_2 \triangleleft l_3.\mathbf{0}, R^{(x_2)}], Q^{(x_1)}\right]$$

Now the exception is thrown, so the process reduces to:

$$\rightharpoonup R^{(x_2)} \parallel Q^{(x_1)}$$

One requirement of our conversion is that we want to keep the exception handlers with their respective default handlers. In our example, this means that the request on b is responsible for triggering its exception handler when an exception is thrown. Now, if we want to model the **throw** using `none`, we would also cancel the exception handler, leaving us unable to reach the situation above. This is why our fragment does not allow disjoint refinement.

The limitation does not have a significant effect on the expressivity of our fragment, because disjoint refinement can be rewritten such that it is allowed. For example, the following process is equivalent to our example with disjoint refinement in (3.1) in terms of reduction:

$$\overline{a}\left[(x_1), \overline{b}[(x_1, x_2), x_1!\langle 5\rangle.\textbf{throw} \parallel x_2?(y).x_2 \lhd l_3.\mathbf{0}, R^{(x_2)} \parallel Q^{(x_1)}], Q^{(x_1)}\right]$$

It is our design choice to make the leftmost process of the parallel composition responsible for invoking the exception handler, so it can cancel the right process with `none`. Processes on the right synchronize with this process using `inl` where there is no throw, and `inr` when there is. See Example 4, Example 5 and Example 6 in Appendix C for a visualization of this process.

Note that the semantics of CYH does not require parallel processes to synchronize. Therefore, reductions such as follows are possible:

$$\begin{aligned}&\overline{a}\left[(x_1, x_2), x_1!\langle 5\rangle.x_1 \lhd l_3.\mathbf{0} \parallel x_2?(y).\textbf{throw}, Q^{(x_1,x_2)}\right]\\ &\rightharpoonup \overline{a}\left[(x_1, x_2), x_1!\langle 5\rangle.x_1 \lhd l_3.\mathbf{0} \parallel \textbf{throw}, Q^{(x_1,x_2)}\right]\\ &\rightharpoonup Q^{(x_1,x_2)}\end{aligned}$$

Here, only a receive on x_2 has happened before the exception is thrown. In our conversion, due to the synchronization we need to perform, the send on x_1 needs to be performed before the exception can be thrown. Different orders of actions might be necessary to model specific situations in which, due to their locality, exception handlers might have access to values received before the exception was thrown. However, at the level of abstraction of the π-calculus, this would be an invisible behaviour, and it is not the intended level of detail for this type of modelling.

Limitation 2: Requests in services

In CYH it is possible to have requests inside services.

$$*a(s)\left[\overline{b}(x_1)\left[(s, x_1), P, R\right], Q\right]$$

In our conversion, when an exception is thrown, the request side cancels the service side with `none`. When its exception handler is invoked, this handler in turn invokes the handlers of the connected services. This is why requests are not allowed in our fragment: a `none` would cancel the request, making it unable to invoke its exception handler.

This limits the ability to model situations in which services need to invoke other services, such as a bank that has to connect to another bank. However, such problems should be easy to overcome. In this banking example, we could, for example, connect two banks using an intermediary request that connects to both banking services.

Limitation 3: Recursion

CYH supports recursion in the form of minimal fixpoints. For example:

$$\mu X.x \triangleright \{l_1 : X, l_2 : x!\langle \texttt{tt} \rangle.\mathbf{0}\}$$

This process repeatedly receives label l_1 until it receives label l_2, after which it outputs `tt` and terminates.

Recursion is still an open field of research for session type theory, especially with a Curry-Howard basis. Since it is not the research topic of this thesis, our fragment does not support recursion, although we do mention it as an option for further research in Section 4.4.

Limitation 4: Conditional statements

The final limitation is that we do not explicitly support conditional statements. The following example uses a conditional statement on a received value to determine which value to send:

$$x?(v).\textbf{if } ok(v) \textbf{ then } P \textbf{ else } Q$$

This process receives a value v. If $ok(v)$ evaluates to a truthy value, the process P gets executes, and otherwise process Q will be run. It is required that P and Q have the same typings for all channels.

We do no have conditional statements in our type system DCPTP. In fact, we do not support conditionals in our fragment at all, since we do not want to clutter our conversion with details that are irrelevant to our research topic. It is trivial to extend the conversion by modelling conditionals using non-deterministic composition, e.g. $P \,\&\, Q$. This is slightly more abstract than the original conditional, since we do not evaluate the condition, but this should not be a problem when one uses the systems discussed in this thesis to study processes on a high level of abstraction.

3.3 The conversion from CYH$^{\text{CH}}$ to DCPTP

In this section we define our conversion from the fragment CYH$^{\text{CH}}$ in Definition 3.1 to our type system DCPTP from Chapter 2. As mentioned in the previous section, our conversion has two important requirements, besides the typing and reduction rules we will discuss in Section 3.4. We have included an example of what a converted process looks like in Appendix B.

The first requirement is that we use `some` and `none` wherever we can, such that we can explore their abilities. The idea is to do this by synchronizing processes at every step of communication, possibly cancelling processes when exceptions are being thrown. The diagram in Figure 3.1 on the following page visualizes this process. It is wise to take a look at Appendix C, in which there are more of these visualizations.

The second requirement is that exception handlers are to be local to their respective default handlers. This is because we want to be able to use these calculi to model real problems. Consider, for example, a model of an internet client/server interaction. When an exception is thrown by the client, the server must invoke its exception handler itself, since the client does not have access to it, let alone have access to the server's resources.

In our definitions, we use a few conventions for notation:

- For each function, a signature is given first. Here we use $\mathcal{P}(-)$ to denote the power set, and $\mathcal{C}$ to denote the set of all channel names.
- We match process terms by referring to the grammar rules in Def. 3.1. This allows us to put restrictions on processes without giving them as an annotation.
- Moreover, because the asynchrony of DCPTP needs fresh channels at almost every communication, the channels in process terms in recursive conversions are renamed where necessary. This is depicted by changing the annotations of the grammar terms. In use with actual process terms, this means that α-conversion needs to be applied to the to be converted process terms (e.g. P becomes $P\{x_3/x\}$ after three steps of synchronization on x).
- Dealing with values and expressions goes beyond the scope of this thesis. Therefore, we use abstract processes to generate and consume expressions and values. Their typing depends on the type of the expression or value in the original process. We do not distinguish between primitive types (e.g. `nat` and `bool`) in CYH and DCPTP, and assume that $\overline{\theta} = \theta$, for any primitive type θ.

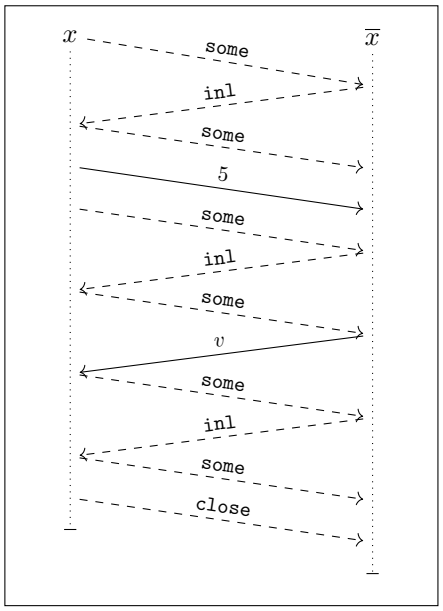


Figure 3.1: Visualization of synchronization in the conversion of $\overline{a}(x)[(x), x!\langle 5\rangle.x?(v).\mathbf{0}, Q^{(x)}]$

The generator process takes an expression e of type θ and outputs it on a fresh channel v.

$$E_{e\to v} \vdash v : \theta$$

The consumer process consumes a value v of type θ.

$$C_v \vdash v : \theta$$

- We write $\prod$ to denote the parallel composition of a process on all channels in a collection of channels, i.e.:

$$\prod_{x\in\{x_1,x_2,\ldots x_n\}} P := P\{x_1/x\} \parallel P\{x_2/x\} \parallel \ldots \parallel P\{x_n/x\}.$$

In the coming pages, we often fill the bottoms of pages with whitespace in order to minimize the breaking of definitions across pages.

As a quick guide to the conversion, we give a short overview. The function C is the main conversion. It converts services into replicated processes, converting their default handlers with $\mathfrak{serv}$, taking care of synchronization, and their exception handlers with $\mathfrak{exc}_\perp$, making sure that $\overline{\texttt{some}}$ can be used to trigger, and $\overline{\texttt{none}}$ to cancel the handler. Requests and refinements are converted with the function $\mathfrak{refine}$, keeping track of connected channels and parallel compositions. It converts default handlers with $\mathfrak{req}$, which takes care of synchronization, and exception handlers with $\mathfrak{exc}_1$, which works similar as for services.

The conversion C is given in Definition 3.2. The conversion uses several different function definitions. We give them one by one, with a general explanation of what they do. Where necessary, we give a more detailed explanation, referring to the specific equation.

(3.2) The conversion of a service creates a channel for its exception handler, which it sends to its connected request. The exception handler can now be invoked with `some`, or cancelled with `none`.

Definition 3.2

$$\mathsf{C} : \mathsf{CYH}^{\mathsf{CH}} \to \mathsf{DCPTP}$$

$$\mathsf{C}\left((\nu a)\left(*a(s)\left[\mathsf{SESSION}^{\emptyset,(s)}_{\checkmark}, \mathsf{SESSION}^{\emptyset,(s)}_{\times}\right] \parallel \mathsf{SERVICES}^{\mathcal{G}\cup(a)}\right)\right) \tag{3.2}$$

$$:= (\nu a)(\,!a(s);$$

$$\quad (\nu s')(\nu s_1)(\overline{s}\langle s', s_1\rangle \parallel$$

$$\quad\quad s'.\texttt{some}_\emptyset(s'_1); \mathfrak{exc}_\perp\left(\mathsf{SESSION}^{\emptyset,(s'_1)}_{\times}\right) \parallel \qquad \text{Def. 3.3: } \mathfrak{exc}$$

$$\quad\quad \mathfrak{serv}\left(\mathsf{SESSION}^{\emptyset,(s_1)}_{\checkmark}\right) \qquad \text{Def. 3.4: } \mathfrak{serv}$$

$$\quad) \parallel$$

$$\quad \mathsf{C}\left(\mathsf{SERVICES}^{\mathcal{G}\cup(a)}\right)$$

$$)$$

$$\mathsf{C}\left(\overline{a}(s)\left[(s), \mathsf{SESSION}^{\mathcal{G},(s)}_{\checkmark}, \mathsf{SESSION}^{\emptyset,(s)}_{\times}\right]\right) \tag{3.3}$$

$$:= \mathfrak{refine}^a_0(\mathcal{G}, \emptyset, \cdot, s, \mathsf{SESSION}^{\mathcal{G},(s)}_{\checkmark}, \mathsf{SESSION}^{\emptyset,(s)}_{\times}) \qquad \text{Def. 3.5: } \mathfrak{refine}$$

$$\mathsf{C}\left(\mathsf{REQUESTS}^{\mathcal{G}} \parallel \mathsf{REQUESTS}^{\mathcal{G}}\right) := \mathsf{C}\left(\mathsf{REQUESTS}^{\mathcal{G}}\right) \parallel \mathsf{C}\left(\mathsf{REQUESTS}^{\mathcal{G}}\right) \tag{3.4}$$

An exception handler is not susceptible to exceptions and cannot throw them, nor make requests. Its conversion $\mathfrak{exc}$, given in Definition 3.3, is therefore straightforward.

Definition 3.3

$$\mathfrak{exc} : \{\mathbf{1}, \bot\} \times \mathcal{P}(\mathcal{C}) \times \mathsf{SESSION}_{\times}^{\emptyset,\mathcal{P}(\mathcal{C})} \to \mathsf{DCPTP}$$

$$\mathfrak{exc}_b(\mathcal{D}, \mathbf{0}) := \begin{cases} \prod_{x\in\mathcal{D}} x.\overline{\texttt{close}} & \text{if } b = \mathbf{1} \\ \prod_{x\in\mathcal{D}} x.\texttt{close} & \text{if } b = \bot \end{cases} \tag{3.5}$$

$$\mathfrak{exc}_b(\mathcal{D}, x?(y).\mathsf{SESSION}_{\times}^{\emptyset,\mathcal{D}}) := x(y, x_1); (C_y \parallel \mathfrak{exc}_b(\mathcal{D}\{x_1/x\}, \mathsf{SESSION}_{\times}^{\emptyset,\mathcal{D}\{x_1/x\}})) \tag{3.6}$$

$$\mathfrak{exc}_b(\mathcal{D}, x!\langle e\rangle.\mathsf{SESSION}_{\times}^{\emptyset,\mathcal{D}}) := (\nu y)(\nu x_1)(\overline{x}\langle y, x_1\rangle \parallel E_{e\to y} \parallel \mathfrak{exc}_b(\mathcal{D}\{x_1/x\}, \mathsf{SESSION}_{\times}^{\emptyset,\mathcal{D}\{x_1/x\}})) \tag{3.7}$$

$$\mathfrak{exc}_b(\mathcal{D}, x \rhd \{l_i : \mathsf{SESSION}_{\times}^{\emptyset,\mathcal{D}}\}_{i\in I}) := x.\texttt{case}(x_1)(\ l_i : \mathfrak{exc}_b(\mathcal{D}\{x_1/x\}, \mathsf{SESSION}_{\times}^{\emptyset,\mathcal{D}\{x_1/x\}})\ )_{i\in I} \tag{3.8}$$

$$\mathfrak{exc}_b(\mathcal{D}, x \lhd l.\mathsf{SESSION}_{\times}^{\emptyset,\mathcal{D}}) := (\nu x_1)(x.\bar{l}\langle x_1\rangle \parallel \mathfrak{exc}_b(\mathcal{D}\{x_1/x\}, \mathsf{SESSION}_{\times}^{\emptyset,\mathcal{D}\{x_1/x\}})) \tag{3.9}$$

$$\mathfrak{exc}_b(\mathcal{D}, \mathsf{SESSION}_{\times}^{\emptyset,\mathcal{D}_1} \parallel \mathsf{SESSION}_{\times}^{\emptyset,\mathcal{D}_2}) := \mathfrak{exc}_b(\mathcal{D}_1, \mathsf{SESSION}_{\times}^{\emptyset,\mathcal{D}_1}) \parallel \mathfrak{exc}_b(\mathcal{D}_2, \mathsf{SESSION}_{\times}^{\emptyset,\mathcal{D}_2}) \tag{3.10}$$

As a convention, we write "$\mathfrak{exc}$" when we mean "$\mathfrak{exc}_{\mathbf{1}}$".

The conversion $\mathfrak{serv}$ of the default handler of a service is given in Definition 3.4. It synchronizes with its connected request before communication. Services cannot make requests, so they only act on single channels. Therefore, a conversion for parallel composition is unnecessary.

(3.11) When an exception is thrown, this is communicated with the request with `inr`.

(3.12)–(3.16) When no exception is thrown, `inl` is used instead.

Definition 3.4

$$\mathfrak{serv} : \mathcal{C} \times \mathsf{SESSION}^{\emptyset,(\mathcal{C})}_{\checkmark} \to \mathsf{DCPTP}$$

$$\mathfrak{serv}(s, \mathbf{throw}) := s.\mathtt{some}_{\emptyset}(s_1); (\nu s_2)(s_1.\overline{\mathtt{inr}}\langle s_2\rangle \parallel \quad (3.11)$$
$$\quad s_2.\mathtt{close}$$
$$)$$

$$\mathfrak{serv}(s, \mathbf{0}) := s.\mathtt{some}_{\emptyset}(s_1); (\nu s_2)(s_1.\overline{\mathtt{inl}}\langle s_2\rangle \parallel \quad (3.12)$$
$$\quad s_2.\mathtt{some}_{\emptyset}(s_3); s_3.\mathtt{close}$$
$$)$$

$$\mathfrak{serv}(s, s?(y).\mathsf{SESSION}^{\emptyset,(s)}_{\checkmark}) := s.\mathtt{some}_{\emptyset}(s_1); (\nu s_2)(s_1.\overline{\mathtt{inl}}\langle s_2\rangle \parallel \quad (3.13)$$
$$\quad s_2.\mathtt{some}_{\emptyset}(s_3); s_3(y, s_4); (C_y \parallel$$
$$\quad\quad \mathfrak{serv}(s_4, \mathsf{SESSION}^{\emptyset,(s_4)}_{\checkmark})$$
$$\quad)$$
$$)$$

$$\mathfrak{serv}(s, s!\langle e\rangle.\mathsf{SESSION}^{\emptyset,(s)}_{\checkmark}) := s.\mathtt{some}_{\emptyset}(s_1); (\nu s_2)(s_1.\overline{\mathtt{inl}}\langle s_2\rangle \parallel \quad (3.14)$$
$$\quad s_2.\mathtt{some}_{\emptyset}(s_3); (\nu y)(\nu s_3)(\overline{s_3}\langle y, s_4\rangle \parallel E_{e\to y} \parallel$$
$$\quad\quad \mathfrak{serv}(s_4, \mathsf{SESSION}^{\emptyset,(s_4)}_{\checkmark})$$
$$\quad)$$
$$)$$

$$\mathfrak{serv}(s, s \rhd \{l_i : \mathsf{SESSION}^{\emptyset,(s)}_{\checkmark}\}_{i\in I}) := s.\mathtt{some}_{\emptyset}(s_1); (\nu s_2)(s_1.\overline{\mathtt{inl}}\langle s_2\rangle \parallel \quad (3.15)$$
$$\quad s_2.\mathtt{some}_{\emptyset}(s_3); s_3.\mathtt{case}(s_4)($$
$$\quad\quad l_i : \mathfrak{serv}(s_4, \mathsf{SESSION}^{\emptyset,(s_4)}_{\checkmark})$$
$$\quad)_{i\in I}$$
$$)$$

$$\begin{aligned}\mathfrak{serv}(s, s \lhd l.\mathsf{SESSION}^{\emptyset,(s)}_{\checkmark}) := s.\mathtt{some}_{\emptyset}(s_1); (\nu s_2)(s_1.\overline{\mathtt{inl}}\langle s_2\rangle \parallel \\ \quad s_2.\mathtt{some}_{\emptyset}(s_3); (\nu s_4)(s_3.\bar{l}\langle s_4\rangle \parallel \\ \quad\quad \mathfrak{serv}(s_4, \mathsf{SESSION}^{\emptyset,(s_4)}_{\checkmark}) \\ \quad) \\ )\end{aligned} \tag{3.16}$$

The conversion $\mathfrak{refine}$ of a request or refinement is given in Definition 3.5. It connects to the service, sets up the exception handler, and runs the default handler.

(3.19)–(3.21) If this is a refinement, there is already an exception handler. It is told to cancel itself, after which the already existing exception handling channels are received for use in the new exception handler. Then, the old exception handler is closed.

(3.22) The exception handling channel for the freshly connected service is received, after which a new exception handling channel is created. It offers two cases: the left case enables invocation of the handler, and the right case forwards exception channels to a new handler. Also, the default handler is put into place.

(3.23)–(3.24) The exception handlers of connected services are invoked, after which the exception handler itself starts.

(3.25)–(3.26) The exception handling channels are forwarded to a new exception handler, after which the current handler is closed.

Definition 3.5

$$\begin{aligned}\mathfrak{refine} : \{0,1,\ldots,6\} \times \mathcal{P}(\mathcal{C}) \cup \mathcal{C} \cup \{\cdot\} \times \mathcal{P}(\mathcal{C}) \times \mathcal{P}(\mathcal{C}) \times \mathcal{C} \cup \{\cdot\} \times \mathcal{C} \cup \{\cdot\} \\ \times \mathsf{SESSION}^{\mathcal{P}(\mathcal{C}),\mathcal{P}(\mathcal{C})}_{\checkmark} \times \mathsf{SESSION}^{\emptyset,\mathcal{P}(\mathcal{C})}_{\times} \to \mathsf{DCPTP}\end{aligned}$$

$$\begin{aligned}&\mathfrak{refine}^{a}_{0}(\mathcal{G}, \mathcal{D}, \eta, s, \mathsf{SESSION}^{\mathcal{G},\mathcal{D}\cup(s)}_{\checkmark}, \mathsf{SESSION}^{\emptyset,\mathcal{D}\cup(s)}_{\times}) \\ &:= (\nu s)(\overline{a}\langle s\rangle \parallel \\ &\quad \mathfrak{refine}^{\cdot}_{1}(\mathcal{G}, \mathcal{D}, \eta, s, \mathsf{SESSION}^{\mathcal{G},\mathcal{D}\cup(s)}_{\checkmark}, \mathsf{SESSION}^{\emptyset,\mathcal{D}\cup(s)}_{\times}) \\ &\quad)\end{aligned} \tag{3.17}$$

$$\begin{aligned}&\mathfrak{refine}^{\cdot}_{1}(\mathcal{G}, \mathcal{D}, \eta, s, \mathsf{SESSION}^{\mathcal{G},\mathcal{D}\cup(s)}_{\checkmark}, \mathsf{SESSION}^{\emptyset,\mathcal{D}\cup(s)}_{\times}) \\ &:= \begin{cases} \mathfrak{refine}^{\cdot}_{2}(\mathcal{G}, \mathcal{D}, y, s, \mathsf{SESSION}^{\mathcal{G},\mathcal{D}\cup(s)}_{\checkmark}, \mathsf{SESSION}^{\emptyset,\mathcal{D}\cup(s)}_{\times}) & \text{if } \eta = y \\ \mathfrak{refine}^{\cdot}_{4}(\mathcal{G}, \mathcal{D}, \cdot, s, \mathsf{SESSION}^{\mathcal{G},\mathcal{D}\cup(s)}_{\checkmark}, \mathsf{SESSION}^{\emptyset,\mathcal{D}\cup(s)}_{\times}) & \text{otherwise (if } \eta = \cdot) \end{cases}\end{aligned} \tag{3.18}$$

$$\mathfrak{refine}^{\cdot}_2(\mathcal{G},\mathcal{D},y,s,\mathsf{SESSION}^{\mathcal{G},\mathcal{D}\cup(s)}_{\checkmark},\mathsf{SESSION}^{\emptyset,\mathcal{D}\cup(s)}_{\times}) \tag{3.19}$$
$$:= (\nu y_1)(y.\overline{\texttt{inr}}\langle y_1\rangle \parallel$$
$$\quad \mathfrak{refine}^{\mathcal{D}}_3(\mathcal{G},\mathcal{D},y_1,s,\mathsf{SESSION}^{\mathcal{G},\mathcal{D}\cup(s)}_{\checkmark},\mathsf{SESSION}^{\emptyset,\mathcal{D}\cup(s)}_{\times})$$
$$)$$

$$\mathfrak{refine}^{(x)\cup\mathcal{D}''}_3(\mathcal{G},\mathcal{D},y,s,\mathsf{SESSION}^{\mathcal{G},\mathcal{D}\cup(s)}_{\checkmark},\mathsf{SESSION}^{\emptyset,\mathcal{D}'\cup(s)}_{\times}) \tag{3.20}$$
$$:= y(x',y_1);\mathfrak{refine}^{\mathcal{D}''}_3(\mathcal{G},\mathcal{D},y_1,s,\mathsf{SESSION}^{\mathcal{G},\mathcal{D}\cup(s)}_{\checkmark},\mathsf{SESSION}^{\emptyset,\mathcal{D}'\{x'/x\}\cup(s)}_{\times})$$

$$\mathfrak{refine}^{\emptyset}_3(\mathcal{G},\mathcal{D},y,s,\mathsf{SESSION}^{\mathcal{G},\mathcal{D}\cup(s)}_{\checkmark},\mathsf{SESSION}^{\emptyset,\mathcal{D}'\cup(s)}_{\times}) \tag{3.21}$$
$$:= y.\overline{\texttt{close}} \parallel \mathfrak{refine}^{\cdot}_4(\mathcal{G},\mathcal{D},\cdot,s,\mathsf{SESSION}^{\mathcal{G},\mathcal{D}\cup(s)}_{\checkmark},\mathsf{SESSION}^{\emptyset,\mathcal{D}'\cup(s)}_{\times})$$

$$\mathfrak{refine}^{\cdot}_4(\mathcal{G},\mathcal{D},\cdot,s,\mathsf{SESSION}^{\mathcal{G},\mathcal{D}\cup(s)}_{\checkmark},\mathsf{SESSION}^{\emptyset,\mathcal{D}'\cup(s)}_{\times}) \tag{3.22}$$
$$:= s(s',s_1);(\nu y)(y.\texttt{case}(y_1)($$
$$\quad\quad y_1.\texttt{some}_{\mathcal{D}'\cup(s')}(y_2);(y_2.\texttt{close} \parallel$$
$$\quad\quad\quad \mathfrak{refine}^{\mathcal{D}'\cup(s')}_5(\emptyset,\emptyset,\cdot,\cdot,\mathbf{0},\mathsf{SESSION}^{\emptyset,\mathcal{D}'\cup(s')}_{\times})$$
$$\quad\quad),$$
$$\quad\quad \mathfrak{refine}^{\mathcal{D}'\cup(s')}_6(\emptyset,\emptyset,y_1,\cdot,\mathbf{0},\mathbf{0})$$
$$\quad) \parallel$$
$$\quad \mathfrak{req}(\mathcal{G},\mathcal{D}\cup(s_1),\emptyset,\cdot,y,\mathsf{SESSION}^{\mathcal{G},\mathcal{D}\cup(s_1)}_{\checkmark}) \qquad \text{Def. 3.6: } \mathfrak{req}$$
$$)$$

$$\mathfrak{refine}^{(x)\cup\mathcal{D}'}_5(\emptyset,\emptyset,\cdot,\cdot,\mathbf{0},\mathsf{SESSION}^{\emptyset,\mathcal{D}}_{\times}) \tag{3.23}$$
$$:= (\nu x_1)(x.\overline{\texttt{some}}\langle x_1\rangle \parallel$$
$$\quad \mathfrak{refine}^{\mathcal{D}'}_5(\emptyset,\emptyset,\cdot,\cdot,\mathbf{0},\mathsf{SESSION}^{\emptyset,\mathcal{D}\{x_1/x\}}_{\times})$$
$$)$$

$$\mathfrak{refine}^{\emptyset}_5(\emptyset,\emptyset,\cdot,\cdot,\mathbf{0},\mathsf{SESSION}^{\emptyset,\mathcal{D}}_{\times}) \tag{3.24}$$
$$:= \mathfrak{exc}(\mathsf{SESSION}^{\emptyset,\mathcal{D}}_{\times}) \qquad \text{Def. 3.3: } \mathfrak{exc}$$

$$\mathfrak{refine}^{(x)\cup\mathcal{D}}_6(\emptyset,\emptyset,y,\cdot,\mathbf{0},\mathbf{0}) \tag{3.25}$$
$$:= (\nu x')(\nu y_1)(\overline{y}\langle x',y_1\rangle \parallel [x \leftrightarrow x'] \parallel$$
$$\quad \mathfrak{refine}^{\mathcal{D}}_6(\emptyset,\emptyset,y_1,\cdot,\mathbf{0},\mathbf{0})$$
$$)$$

$$\mathfrak{refine}^{\emptyset}_6(\emptyset,\emptyset,y,\cdot,\mathbf{0},\mathbf{0}) \tag{3.26}$$
$$:= y.\texttt{close}$$

The conversion of a request's default handler $\mathfrak{req}$ is given in Definition 3.6. It delegates the conversion to different functions depending on the process.

(3.32) The amount of communications of parallel processes is calculated, taking the maximum of the two as the lifetime for the composition. A new parallelism channel is created, which becomes a slave (Λ for $\delta o \tilde{\upsilon} \Lambda o \varsigma$, which is ancient Greek for "slave") in the left process, and a master (μ for Master) in the right.

Definition 3.6

$$\mathfrak{req} : \mathcal{P}(\mathcal{C}) \times \mathcal{P}(\mathcal{C}) \times \mathcal{P}((\mathcal{C} \times \mathbb{N}_0)) \times (\mathcal{C} \times \mathbb{N}_0) \cup \{\cdot\} \times \mathcal{C} \cup \{\cdot\} \times \mathsf{SESSION}_{\checkmark}^{\mathcal{P}(\mathcal{C}),\mathcal{P}(\mathcal{C})} \to \mathsf{DCPTP}$$

$$\mathfrak{req}(\mathcal{G}, \mathcal{D}, \Lambda, \mu, \eta, \mathbf{0}) \tag{3.27}$$
$$:= \mathfrak{reqstep}_0^{\cdot}(\mathcal{G}, \mathcal{D}, \Lambda, \mu, \eta, \cdot, \mathbf{0}) \quad \text{Def. 3.7: } \mathfrak{reqstep}$$

$$\mathfrak{req}(\mathcal{G}, \mathcal{D}, \Lambda, \mu, \eta, x?(y).\mathsf{SESSION}_{\checkmark}^{\mathcal{G},\mathcal{D}}) \tag{3.28}$$
$$:= \mathfrak{reqstep}_0^{\cdot}(\mathcal{G}, \mathcal{D}, \Lambda, \mu, \eta, x, x?(y).\mathsf{SESSION}_{\checkmark}^{\mathcal{G},\mathcal{D}})$$

$$\mathfrak{req}(\mathcal{G}, \mathcal{D}, \Lambda, \mu, \eta, x!\langle e\rangle.\mathsf{SESSION}_{\checkmark}^{\mathcal{G},\mathcal{D}}) \tag{3.29}$$
$$:= \mathfrak{reqstep}_0^{\cdot}(\mathcal{G}, \mathcal{D}, \Lambda, \mu, \eta, x, x!\langle e\rangle.\mathsf{SESSION}_{\checkmark}^{\mathcal{G},\mathcal{D}})$$

$$\mathfrak{req}(\mathcal{G}, \mathcal{D}, \Lambda, \mu, \eta, x \rhd \{l_i : \mathsf{SESSION}_{\checkmark}^{\mathcal{G},\mathcal{D}}\}_{i\in I}) \tag{3.30}$$
$$:= \mathfrak{reqstep}_0^{\cdot}(\mathcal{G}, \mathcal{D}, \Lambda, \mu, \eta, x, x \rhd \{l_i : \mathsf{SESSION}_{\checkmark}^{\mathcal{G},\mathcal{D}}\}_{i\in I})$$

$$\mathfrak{req}(\mathcal{G}, \mathcal{D}, \Lambda, \mu, \eta, x \lhd l.\mathsf{SESSION}_{\checkmark}^{\mathcal{G},\mathcal{D}}) \tag{3.31}$$
$$:= \mathfrak{reqstep}_0^{\cdot}(\mathcal{G}, \mathcal{D}, \Lambda, \mu, \eta, x, x \lhd l.\mathsf{SESSION}_{\checkmark}^{\mathcal{G},\mathcal{D}})$$

$$\mathfrak{req}(\mathcal{G}, \mathcal{D}, \Lambda, \mu, \eta, \mathsf{SESSION}_{\checkmark}^{\emptyset,\mathcal{D}_1} \parallel \mathsf{SESSION}_{\checkmark}^{\emptyset,\mathcal{D}_2}) \tag{3.32}$$
$$:= (\nu z)($$
$$\quad \mathfrak{req}(\emptyset, \mathcal{D}_1, \Lambda \cup (z, \mathfrak{card}(\mathsf{SESSION}^{\emptyset,\mathcal{D}_1} \parallel \mathsf{SESSION}^{\emptyset,\mathcal{D}_2})), \mu, \eta, \mathsf{SESSION}^{\emptyset,\mathcal{D}_1}) \parallel \quad \text{Def. 3.8: } \mathfrak{card}$$
$$\quad \mathfrak{req}(\emptyset, \mathcal{D}_2, \emptyset, (z, \mathfrak{card}(\mathsf{SESSION}^{\emptyset,\mathcal{D}_1} \parallel \mathsf{SESSION}^{\emptyset,\mathcal{D}_2})), \cdot, \mathsf{SESSION}^{\emptyset,\mathcal{D}_2})$$
$$)$$

$$\mathfrak{req}(\mathcal{G}, \mathcal{D}, \emptyset, \cdot, y, \overline{a}(s)[\mathcal{D} \cup (s), \mathsf{SESSION}_{\checkmark}^{\mathcal{G},\mathcal{D}\cup(s)}, \mathsf{SESSION}_{\times}^{\emptyset,\mathcal{D}\cup(s)}]) \tag{3.33}$$
$$:= \mathfrak{refine}_0^a(\mathcal{G}, \mathcal{D}, y, s, \mathsf{SESSION}_{\checkmark}^{\mathcal{G},\mathcal{D}\cup(s)}, \mathsf{SESSION}_{\times}^{\emptyset,\mathcal{D}\cup(s)}) \quad \text{Def. 3.5: } \mathfrak{refine}$$

$$\mathfrak{req}(\mathcal{G}, \mathcal{D}, \Lambda, \mu, \eta, \textbf{throw}) \tag{3.34}$$
$$:= \mathfrak{throw}_0(\mathcal{D}, \Lambda, \mu, \eta) \quad \text{Def. 3.9: } \mathfrak{throw}$$

The conversion of a communication in the default handler of a request $\mathfrak{reqstep}$ is given in Definition 3.7. It synchronizes with the connected service, as well as with slaves and master (the parallelism channels; see (3.32)).

(3.35) If there is a master, listen to it for an exception, possibly forwarding a `none` to all connected services and slaves.

(3.36)–(3.37) If the process is $\mathbf{0}$, we need to wait for the other sides of parallel compositions to finish. When they are all done, we synchronise with all connected services. Otherwise, we only synchronise with the relevant service. When an exception is received, we only forward it to the other connected services and the slaves and master, and we invoke the exception handler if it is present.

(3.38)–(3.39) We synchronise with slaves that are not done yet. When the process is not $\mathbf{0}$, none of them are done yet. Otherwise, when they are all done, we synchronise with them all, so we can close all channels. When an exception is received, we only forward it to the other slaves and the connected services and master, and we invoke the exception handler if it is present.

(3.40) If there is a master, we tell it that we are not throwing an exception, and listen to if for an exception again.

(3.41) If the process is $\mathbf{0}$ and not all parallel processes are done yet, we need to send `some` to those slaves that are still active (3.42)–(3.43). If all are done, we send `some` to all connected services and slaves, and close all those channels, as well as the master chancel. Also, we cancel the exception handler. If the process is not $\mathbf{0}$, the actual communication can occur (3.44)–(3.47).

Definition 3.7

$$\begin{aligned}\mathfrak{reqstep} : \{0,1,\ldots,5\} \times \mathcal{P}(\mathcal{C}) \cup \mathcal{P}((\mathcal{C}\times\mathbb{N}_0)) \cup \{\cdot\} \times \mathcal{P}(\mathcal{C}) \times \mathcal{P}(\mathcal{C}) \times \mathcal{P}((\mathcal{C}\times\mathbb{N}_0)) \\ \times(\mathcal{C}\times\mathbb{N}_0) \cup \{\cdot\} \times \mathcal{C} \cup \{\cdot\} \times \mathcal{C} \cup \{\cdot\} \times \mathsf{SESSION}_{\checkmark}^{\mathcal{P}(\mathcal{C}),\mathcal{P}(\mathcal{C})} \to \mathsf{DCPTP}\end{aligned}$$

$$\mathfrak{reqstep}^{\cdot}_0(\mathcal{G},\mathcal{D},\Lambda,\mu,\eta,\delta,\mathsf{SESSION}_{\checkmark}^{\mathcal{G},\mathcal{D}}) \tag{3.35}$$

$$:= \begin{cases} x.\mathtt{some}_{\mathcal{D}\cup\mathfrak{chans}(\Lambda)}(x_1);\mathfrak{reqstep}^{\cdot}_1(\mathcal{G},\mathcal{D},\Lambda,(x_1,n),\cdot,\delta,\mathsf{SESSION}_{\checkmark}^{\mathcal{G},\mathcal{D}}) & \text{if } \mu=(x,n) \\ \mathfrak{reqstep}^{\cdot}_1(\mathcal{G},\mathcal{D},\Lambda,\cdot,\eta,\delta,\mathsf{SESSION}_{\checkmark}^{\mathcal{G},\mathcal{D}}) & \text{otherwise (if } \mu=\cdot) \end{cases}$$

Def. 3.11: $\mathfrak{chans}$

$$\mathfrak{reqstep}^{\cdot}_1(\mathcal{G},\mathcal{D},\Lambda,\mu,\eta,\delta,\mathsf{SESSION}_{\checkmark}^{\mathcal{G},\mathcal{D}}) \tag{3.36}$$

$$:= \begin{cases} \mathfrak{reqstep}_2^{(x)}(\mathcal{G},\mathcal{D},\Lambda,\mu,\eta,x,\mathsf{SESSION}_{\checkmark}^{\mathcal{G},\mathcal{D}}) & \text{if } \delta=x \\ \mathfrak{reqstep}_2^{\mathcal{D}}(\mathcal{G},\mathcal{D},\Lambda,\mu,\eta,\cdot,\mathbf{0}) & \text{if } \delta=\cdot \text{ and } (\Lambda\neq\emptyset \implies \mathfrak{getlive}(\Lambda)=\emptyset) \\ & \text{and not } \mathfrak{islive}(\mu) \\ \mathfrak{reqstep}_2^{\emptyset}(\mathcal{G},\mathcal{D},\Lambda,\mu,\eta,\cdot,\mathbf{0}) & \text{otherwise} \end{cases}$$

Def. 3.13: $\mathfrak{getlive}$

Def. 3.14: $\mathfrak{islive}$

$$\mathfrak{reqstep}_2^{(x)\cup\mathcal{D}'}(\mathcal{G},\mathcal{D},\Lambda,\mu,\eta,\delta,\mathsf{SESSION}_{\checkmark}^{\mathcal{G},\mathcal{D}}) \tag{3.37}$$

$$:= (\nu x_1)(x.\overline{\texttt{some}}\langle x_1\rangle \parallel$$
$$\quad x_1.\texttt{case}(x_2)($$
$$\quad\quad \mathfrak{reqstep}_2^{\mathcal{D}'}(\mathcal{G},\mathcal{D}\{x_2/x\},\Lambda,\mu,\eta,\mathfrak{rename}(\delta,x,x_2),\mathsf{SESSION}_{\checkmark}^{\mathcal{G},\mathcal{D}\{x_2/x\}}),$$ Def. 3.12: $\mathfrak{rename}$
$$\quad\quad x_2.\overline{\texttt{close}} \parallel \mathfrak{throw}_0(\mathcal{D}\setminus x,\Lambda,\cdot,\eta) \parallel \mathfrak{throwmaster}(\mu)$$ Def. 3.9: $\mathfrak{throw}$
$$\quad)$$ Def. 3.10: $\mathfrak{throwmaster}$
$$)$$

$$\mathfrak{reqstep}_2^{\emptyset}(\mathcal{G},\mathcal{D},\Lambda,\mu,\eta,\delta,\mathsf{SESSION}_{\checkmark}^{\mathcal{G},\mathcal{D}}) \tag{3.38}$$

$$:= \begin{cases} \mathfrak{reqstep}_3^{\mathfrak{getlive}(\Lambda)}(\mathcal{G},\mathcal{D},\Lambda,\mu,\eta,\delta,\mathsf{SESSION}_{\checkmark}^{\mathcal{G},\mathcal{D}}) & \text{if } \delta\neq\cdot \text{ or} \\ & (\Lambda\neq\emptyset \implies \mathfrak{getlive}(\Lambda)\neq\emptyset) \text{ or } \mathfrak{islive}(\mu) \\ \mathfrak{reqstep}_3^{\Lambda}(\mathcal{G},\mathcal{D},\Lambda,\mu,\eta,\delta,\mathbf{0}) & \text{otherwise} \end{cases}$$ Def. 3.13: $\mathfrak{getlive}$ Def. 3.14: $\mathfrak{islive}$

$$\mathfrak{reqstep}_3^{(x,n)\cup\Lambda'}(\mathcal{G},\mathcal{D},\Lambda,\mu,\eta,\delta,\mathsf{SESSION}_{\checkmark}^{\mathcal{G},\mathcal{D}}) \tag{3.39}$$

$$:= (\nu x_1)(x.\overline{\texttt{some}}\langle x_1\rangle \parallel$$
$$\quad x_1.\texttt{case}(x_2)($$
$$\quad\quad \mathfrak{reqstep}_3^{\Lambda'}(\mathcal{G},\mathcal{D},\Lambda\{x_2/x\},\mu,\eta,\delta,\mathsf{SESSION}_{\checkmark}^{\mathcal{G},\mathcal{D}}),$$
$$\quad\quad x_2.\overline{\texttt{close}} \parallel \mathfrak{throw}_0(\mathcal{D},\Lambda\setminus x,\cdot,\eta) \parallel \mathfrak{throwmaster}(\mu)$$
$$\quad)$$
$$)$$

$$\mathfrak{reqstep}_3^{\emptyset}(\mathcal{G},\mathcal{D},\Lambda,\mu,\eta,\delta,\mathsf{SESSION}_{\checkmark}^{\mathcal{G},\mathcal{D}}) \tag{3.40}$$

$$:= \begin{cases} (\nu x_1)(& \text{if } \mu=(x,n) \\ \quad x.\overline{\texttt{inl}}\langle x_1\rangle \parallel & \\ \quad x_1.\texttt{some}_{\mathcal{D}\cup\mathfrak{chans}(\Lambda)}(x_2);\mathfrak{reqstep}_4^{\cdot}(\mathcal{G},\mathcal{D},\Lambda,(x_2,n),\cdot,\delta,\mathsf{SESSION}_{\checkmark}^{\mathcal{G},\mathcal{D}}) & \\ ) & \\ \mathfrak{reqstep}_4^{\cdot}(\mathcal{G},\mathcal{D},\Lambda,\cdot,\eta,\delta,\mathsf{SESSION}_{\checkmark}^{\mathcal{G},\mathcal{D}}) & \text{otherwise (if } \mu=\cdot) \end{cases}$$ Def. 3.11: $\mathfrak{chans}$

$$\mathfrak{reqstep}^{\cdot}_4(\mathcal{G},\mathcal{D},\Lambda,\mu,\eta,\delta,\mathsf{SESSION}^{\mathcal{G},\mathcal{D}}_{\checkmark}) \tag{3.41}$$

$$:= \begin{cases} (\nu x_1)(& \text{if } \delta = x \\ \quad x.\overline{\texttt{some}}\langle x_1\rangle \parallel & \\ \quad \mathfrak{reqstep}^{\Lambda}_5(\mathcal{G},\mathcal{D}\{x_1/x\},\Lambda,\mu,\eta,x_1,\mathsf{SESSION}^{\mathcal{G},\mathcal{D}\{x_1/x\}}_{\checkmark}) & \\ ) & \\ \mathfrak{reqstep}^{\mathfrak{getlive}(\Lambda)}_5(\mathcal{G},\mathcal{D},\Lambda,\mu,\eta,\cdot,\mathbf{0}) & \text{if } \delta = \cdot \text{ and } ((\Lambda \neq \emptyset \implies \mathfrak{getlive}(\Lambda) \neq \emptyset) \text{ or } \mathfrak{islive}(\mu)) \\ \prod_{x\in\mathcal{D}\cup\mathfrak{chans}(\Lambda)} (\nu x_1)\left(x.\overline{\texttt{some}}\langle x_1\rangle \parallel x_1.\overline{\texttt{close}}\right) \parallel & \text{otherwise} \\ \mathfrak{closemaster}(\mu) \parallel \mathfrak{cancel}(\eta) & \end{cases}$$

Def. 3.13: $\mathfrak{getlive}$
Def. 3.14: $\mathfrak{islive}$
Def. 3.11: $\mathfrak{chans}$
Def. 3.15: $\mathfrak{closemaster}$
Def. 3.16: $\mathfrak{cancel}$

$$\mathfrak{reqstep}^{(x,n)\cup\Lambda'}_5(\mathcal{G},\mathcal{D},\Lambda,\mu,\eta,\delta,\mathsf{SESSION}^{\mathcal{G},\mathcal{D}}_{\checkmark}) \tag{3.42}$$
$$:= (\nu x_1)(x.\overline{\texttt{some}}\langle x_1\rangle \parallel \quad \mathfrak{reqstep}^{\Lambda'}_5(\mathcal{G},\mathcal{D},\Lambda\{x_1/x\},\mu,\eta,\delta,\mathsf{SESSION}^{\mathcal{G},\mathcal{D}}_{\checkmark}) \ )$$

$$\mathfrak{reqstep}^{\emptyset}_5(\mathcal{G},\mathcal{D},\Lambda,\mu,\eta,\cdot,\mathbf{0}) \tag{3.43}$$
$$:= \mathfrak{reqstep}^{\cdot}_0(\mathcal{G},\mathcal{D},\mathfrak{nextstep}(\Lambda),\mathfrak{nextstepmaster}(\mu),\eta,\cdot,\mathbf{0})$$

Def. 3.17: $\mathfrak{nextstep}$
Def. 3.18: $\mathfrak{nextstepmaster}$

$$\mathfrak{reqstep}^{\emptyset}_5(\mathcal{G},\mathcal{D},\Lambda,\mu,\eta,x,x?(y).\mathsf{SESSION}^{\mathcal{G},\mathcal{D}}_{\checkmark}) \tag{3.44}$$
$$:= x(y,x_1);(C_y \parallel \quad \mathfrak{req}(\mathcal{G},\mathcal{D}\{x_1/x\},\mathfrak{nextstep}(\Lambda),\mathfrak{nextstepmaster}(\mu),\eta,\mathsf{SESSION}^{\mathcal{G},\mathcal{D}\{x_1/x\}}_{\checkmark}) \ )$$

$$\mathfrak{reqstep}^{\emptyset}_5(\mathcal{G},\mathcal{D},\Lambda,\mu,\eta,x,x!\langle e\rangle.\mathsf{SESSION}^{\mathcal{G},\mathcal{D}}_{\checkmark}) \tag{3.45}$$
$$:= (\nu y)(\nu x_1)(\overline{x}\langle y,x_1\rangle \parallel E_{e\to y} \parallel \quad \mathfrak{req}(\mathcal{G},\mathcal{D}\{x_1/x\},\mathfrak{nextstep}(\Lambda),\mathfrak{nextstepmaster}(\mu),\eta,\mathsf{SESSION}^{\mathcal{G},\mathcal{D}\{x_1/x\}}_{\checkmark}) \ )$$

$$\mathfrak{reqstep}^{\emptyset}_5(\mathcal{G},\mathcal{D},\Lambda,\mu,\eta,x,x \rhd \{l_i : \mathsf{SESSION}^{\mathcal{G},\mathcal{D}}_{\checkmark}\}_{i\in I}) \tag{3.46}$$
$$:= x.\texttt{case}(x_1)(\quad l_i : \mathfrak{req}(\mathcal{G},\mathcal{D}\{x_1/x\},\mathfrak{nextstep}(\Lambda),\mathfrak{nextstepmaster}(\mu),\eta,\mathsf{SESSION}^{\mathcal{G},\mathcal{D}\{x_1/x\}}_{\checkmark}) \ )_{i\in I}$$

$$\mathfrak{reqstep}^{\emptyset}_5(\mathcal{G},\mathcal{D},\Lambda,\mu,\eta,x,x \lhd l.\mathsf{SESSION}^{\mathcal{G},\mathcal{D}}_{\checkmark}) \tag{3.47}$$
$$:= (\nu x_1)(x.\bar{l}\langle x_1\rangle \parallel \quad \mathfrak{req}(\mathcal{G},\mathcal{D}\{x_1/x\},\mathfrak{nextstep}(\Lambda),\mathfrak{nextstepmaster}(\mu),\eta,\mathsf{SESSION}^{\mathcal{G},\mathcal{D}\{x_1/x\}}_{\checkmark}) \ )$$

Definition 3.8 (Cardinality (length) of processes)

$$\mathfrak{card} : \mathsf{SESSION}^{\emptyset,\mathcal{P}(\mathcal{C})}_{\checkmark} \to \mathbb{N}_0$$

$$\mathfrak{card}(\mathbf{0}) := 0$$

$$\mathfrak{card}(\mathbf{throw}) := 0$$

$$\mathfrak{card}(\mathsf{SESSION}^{\emptyset,\mathcal{D}_1}_{\checkmark} \parallel \mathsf{SESSION}^{\emptyset,\mathcal{D}_2}_{\checkmark} := \max(\mathfrak{card}(\mathsf{SESSION}^{\emptyset,\mathcal{D}_1}_{\checkmark}), \mathfrak{card}(\mathsf{SESSION}^{\emptyset,\mathcal{D}_2}_{\checkmark}))$$

$$\mathfrak{card}(x?(y).\mathsf{SESSION}^{\emptyset,\mathcal{D}}_{\checkmark}) := 1 + \mathfrak{card}(\mathsf{SESSION}^{\emptyset,\mathcal{D}}_{\checkmark})$$

$$\mathfrak{card}(x!\langle e\rangle.\mathsf{SESSION}^{\emptyset,\mathcal{D}}_{\checkmark}) := 1 + \mathfrak{card}(\mathsf{SESSION}^{\emptyset,\mathcal{D}}_{\checkmark})$$

$$\mathfrak{card}(x \rhd \{l_i : \mathsf{SESSION}^{\emptyset,\mathcal{D}}_{\checkmark}\}_{i\in I}) := 1 + \max(\{\mathfrak{card}(\mathsf{SESSION}^{\emptyset,\mathcal{D}}_{\checkmark})\}_{i\in I})$$

$$\mathfrak{card}(x \lhd l.\mathsf{SESSION}^{\emptyset,\mathcal{D}}_{\checkmark}) := 1 + \mathfrak{card}(\mathsf{SESSION}^{\emptyset,\mathcal{D}}_{\checkmark})$$

When an exception is thrown, the $\mathfrak{throw}$ function, given in Definition 3.9, cancels connected requests and slaves, forwards the exception to the master, and invokes the exception handler.

Definition 3.9

$$\mathfrak{throw} : \{0,1,\ldots,4\} \times \mathcal{P}(\mathcal{C}) \times \mathcal{P}((\mathcal{C}\times\mathbb{N}_0)) \times (\mathcal{C}\times\mathbb{N}_0) \cup \{\cdot\} \times \mathcal{C} \cup \{\cdot\} \to \mathsf{DCPTP}$$

$$\mathfrak{throw}_0(\mathcal{D},\Lambda,\mu,\eta) := \begin{cases} \mathfrak{throw}_1(\mathcal{D},\Lambda,\cdot,\eta) \parallel \mathfrak{throw}_3(\emptyset,\emptyset,(x,n),\cdot) & \text{if } \mu = (x,n) \\ \mathfrak{throw}_1(\mathcal{D},\Lambda,\cdot,\eta) & \text{otherwise (if } \mu = \cdot) \end{cases}$$

$$\mathfrak{throw}_1(\mathcal{D},\Lambda,\cdot,\eta) := \begin{cases} \mathfrak{throw}_2(\mathcal{D},\Lambda,\cdot,\cdot) \parallel \mathfrak{throw}_4(\emptyset,\emptyset,\cdot,x) & \text{if } \eta = x \\ \mathfrak{throw}_2(\mathcal{D},\Lambda,\cdot,\cdot) & \text{otherwise (if } \eta = \cdot) \end{cases}$$

$$\mathfrak{throw}_2(\mathcal{D},\Lambda,\cdot,\cdot) := \prod_{x\in\mathcal{D}} x.\overline{\texttt{none}} \parallel \prod_{(x,n)\in\Lambda} x.\overline{\texttt{none}}$$

$$\mathfrak{throw}_3(\emptyset,\emptyset,(x,n),\cdot) := x.\texttt{some}_{\emptyset}(x_1); \mathfrak{throwmaster}(x_1)$$

Def. 3.10: $\mathfrak{throwmaster}$

$$\begin{aligned} \mathfrak{throw}_4(\emptyset,\emptyset,\cdot,x) := {} & (\nu x_1)(\\ & \quad x.\overline{\texttt{inl}}\langle x_1\rangle \parallel \\ & \quad (\nu x_2)(\\ & \quad\quad x_1.\overline{\texttt{some}}\langle x_2\rangle \parallel \\ & \quad\quad x_2.\overline{\texttt{close}} \\ & \quad) \\ &) \end{aligned}$$

Definition 3.10

$$\mathfrak{throwmaster} : \mathcal{C} \cup \{\cdot\} \to \mathsf{DCPTP}$$

$$\mathfrak{throwmaster}(x) := (\nu x_1)(x.\overline{\texttt{inr}}\langle x_1\rangle \parallel \\ \quad x_1.\texttt{close} \\ )$$

$$\mathfrak{throwmaster}(\cdot) := \mathbf{0}$$

The auxiliary function $\mathfrak{chans}$, given in Definition 3.11, takes the list of slave channels and removes their length annotation.

Definition 3.11

$$\mathfrak{chans} : \mathcal{P}((\mathcal{C} \times \mathbb{N}_0)) \to \mathcal{P}(\mathcal{C})$$

$$\mathfrak{chans}((x, n) \cup \Lambda) := (x) \cup \mathfrak{chans}(\Lambda)$$

$$\mathfrak{chans}(\emptyset) := \emptyset$$

The auxiliary function $\mathfrak{rename}$, given in Definition 3.12, renames a specific channel, and leaves it unchanged if it does not match or is $\cdot$.

Definition 3.12

$$\mathfrak{rename} : \mathcal{C} \cup \{\cdot\} \times \mathcal{C} \times \mathcal{C} \to \mathcal{C} \cup \{\cdot\}$$

$$\mathfrak{rename}(\delta, x, x') := \begin{cases} x' & \text{if } \delta = x \\ \delta & \text{otherwise} \end{cases}$$

The auxiliary function $\mathfrak{getlive}$, given in Definition 3.13, gives a list of those slave channels that belong to a parallel process that is not finished yet. The auxiliary predicate $\mathfrak{islive}$, given in Definition 3.14, checks if the parallel process belonging to the master channel is not finished yet

Definition 3.13

$$\mathfrak{getlive} : \mathcal{P}((\mathcal{C} \times \mathbb{N}_0)) \to \mathcal{P}((\mathcal{C} \times \mathbb{N}_0))$$

$$\mathfrak{getlive}((x, n{+}1) \cup \Lambda) := (x, n{+}1) \cup \mathfrak{getlive}(\Lambda)$$

$$\mathfrak{getlive}((x, 0) \cup \Lambda) := \mathfrak{getlive}(\Lambda)$$

$$\mathfrak{getlive}(\emptyset) := \emptyset$$

Definition 3.14

$$\mathfrak{islive} : (\mathcal{C} \times \mathbb{N}_0) \cup \{\cdot\} \to \mathbb{B}$$
$$\mathfrak{islive}((x, n{+}1)) := \top$$
$$\mathfrak{islive}((x, 0)) := \bot$$
$$\mathfrak{islive}(\cdot) := \bot$$

Definition 3.15

$$\mathfrak{closemaster} : (\mathcal{C} \times \mathbb{N}_0) \cup \{\cdot\} \to \mathsf{DCPTP}$$
$$\mathfrak{closemaster}((x, n)) := x.\mathtt{close}$$
$$\mathfrak{closemaster}(\cdot) := \mathbf{0}$$

Definition 3.16

$$\mathfrak{cancel} : \mathcal{C} \cup \{\cdot\}$$
$$\mathfrak{cancel}(x) := (\nu x_1)(x.\overline{\mathtt{inl}}\langle x_1\rangle \parallel \begin{array}{l} \\ \vdots \\ \\ \end{array} \begin{array}{l} \\ x_1.\overline{\mathtt{none}} \\ \\ \end{array})$$
$$\mathfrak{cancel}(\cdot) := \mathbf{0}$$

The auxiliary function $\mathfrak{nextstep}$, given in Definition 3.17, decreases all the counters in a list of slave channels that are bigger than one. The auxiliary function $\mathfrak{nextstepmaster}$, given in Definition 3.18, decreases the counter of the master channel if it is bigger than one.

Definition 3.17

$$\mathfrak{nextstep} : \mathcal{P}((\mathcal{C} \times \mathbb{N}_0)) \to \mathcal{P}((\mathcal{C} \times \mathbb{N}_0))$$
$$\mathfrak{nextstep}((x, n{+}1) \cup \Lambda) := (x, n) \cup \mathfrak{nextstep}(\Lambda)$$
$$\mathfrak{nextstep}((x, 0) \cup \Lambda) := (x, 0) \cup \mathfrak{nextstep}(\Lambda)$$
$$\mathfrak{nextstep}(\emptyset) := \emptyset$$

Definition 3.18

$$\mathfrak{nextstepmaster} : (\mathcal{C} \times \mathbb{N}_0) \cup \{\cdot\} \to (\mathcal{C} \times \mathbb{N}_0) \cup \{\cdot\}$$
$$\mathfrak{nextstepmaster}((x, n{+}1)) := (x, n)$$
$$\mathfrak{nextstepmaster}((x, 0)) := (x, 0)$$
$$\mathfrak{nextstepmaster}(\cdot) := \cdot$$

3.4 Results

In this section we formalise the connection between processes in $\mathsf{CYH}^{\mathsf{CH}}$ and their conversions into DCPTP, in terms of types in Section 3.4.1 and in terms of deadlock-freeness in Section 3.4.2. Moreover, we conjecture a connection in terms of reductions in Section 3.4.3

3.4.1 Typing results

In order to apply results that follow from the Curry-Howard correspondence between classical linear logic and our process calculus DCPTP, we need to show that the processes that result from our conversion are well-typed. This means that there are no free channels, i.e. that all channels are connected to two processes of dual behaviour.

Conversion of types

Our proof of well-typedness involves several lemmas that show other interesting results about the connection between processes in $\mathsf{CYH}^{\mathsf{CH}}$ and their conversions. They show a correspondence in the types of channels within services, requests, and exception handlers. For this, we need to be able to transform $\mathsf{CYH}^{\mathsf{CH}}$ types into DCPTP types. As stated at the beginning of Section 3.3, we assume no distinction between primitive types θ in either system.

Definition 3.19

$$\mathcal{T}_b(\downarrow(\theta).\alpha) := \theta \parr \mathcal{T}_b(\alpha)$$
$$\mathcal{T}_b(\uparrow(\theta).\alpha) := \theta \otimes \mathcal{T}_b(\alpha)$$
$$\mathcal{T}_b(\oplus\{l_i : \alpha_i\}_{i\in I}) := \oplus\{l_i : \mathcal{T}_b(\alpha_i)\}_{i\in I}$$
$$\mathcal{T}_b(\&\{l_i : \alpha_i\}_{i\in I}) := \&\{l_i : \mathcal{T}_b(\alpha_i)\}_{i\in I}$$
$$\mathcal{T}_b(\alpha\{\![\beta]\!\}) := \mathcal{T}_b(\alpha)$$
$$\mathcal{T}_{\mathbf{1}}(\mathtt{end}) := \mathbf{1}$$
$$\mathcal{T}_{\bot}(\mathtt{end}) := \bot$$

As a convention, we write "$\mathcal{T}$" when we mean "$\mathcal{T}_{\mathbf{1}}$".

Types of requests

Next, we need a way of expressing the type of a channel in the default handler of a request. This shows all the points in a step of communication where an exception might be thrown. Only after a `some`, then an `inl`, followed by another `some` can the communication take place.

Definition 3.20

$$
\begin{aligned}
?\mathbf{1} &:= \&(\&\mathbf{1} \,\&\, \mathbf{1}) \\
?(A \otimes B) &:= \&(\&(A \otimes ?B) \,\&\, \mathbf{1}) \\
?(A \parr B) &:= \&(\&(A \parr ?B) \,\&\, \mathbf{1}) \\
?(A \oplus B) &:= \&(\&(?A \oplus ?B) \,\&\, \mathbf{1}) \\
?(A \,\&\, B) &:= \&(\&(?A \& ?B) \,\&\, \mathbf{1})
\end{aligned}
$$

Since converted requests are self-contained (i.e. they are in control of their own default and exception handlers), upon connection with a service, a (cancelable) channel is received over which communication can take place after an exception has been thrown. This is included in the full type of the request.

Definition 3.21

$$A?E := (\&E) \parr ?A$$

Auxiliary types

Moreover, when refinement takes place, the new exception handler receives the handling channels from the current one. The following definition expresses this process, where $\mathcal{B}$ is a list of types.

Definition 3.22

$$
\begin{aligned}
\bullet\emptyset &:= \mathbf{1} \\
\bullet((A) \cup \mathcal{B}) &:= A \parr (\bullet\mathcal{B})
\end{aligned}
$$

Our final definition describes the type of a slave channel, used for parallelism. It is very similar to the default handler types in Def. 3.20, except that no actual communication takes place.

Definition 3.23

$$
\begin{aligned}
\|_0 &:= ?\mathbf{1} \\
\|_{n+1} &:= \&(\&\|_n \,\&\, \mathbf{1})
\end{aligned}
$$

The results

Our main well-typedness result is given in Theorem 3.1. Lemma 3.7 shows the connection between the types of default handlers in requests, with Lemma 3.6 showing the same connection, but including the exception handler's type. Lemma 3.3 shows a similar connection for the server's side of things.

In our proofs, we often omit typings that remain unchanged across steps. Moreover, unless otherwise stated, we superscript lists of channels (e.g. $\mathcal{D}$ or Λ) or optional channels (e.g. $\mu \in \mathcal{P}(\mathcal{C}) \cup \{\cdot\}$) with a number to denote the subscription of matching channels with that number. For example, $\mathcal{D}^3 = \mathcal{D}\{x_3/x\}_{x \in \mathcal{D}}$. This is to improve readability, especially when new channel names are required due to asynchrony.

Theorem 3.1

Given some well-typed program $P \in \mathsf{CYH}^{\mathsf{CH}}$, the converted process $\mathsf{C}(P)$ is well-typed, i.e. $\mathsf{C}(P) \vdash \cdot; \cdot$, with C as in Def. 3.2.

Proof. By the grammar in Def. 3.1, it must be that $P \in \mathsf{SERVICES}^{\emptyset}$. Hence, the thesis follows from Lemma 3.2. □

Lemma 3.2

Take some $\Gamma \in \mathcal{P}(\mathcal{C})$, and some $P \in \mathsf{SERVICES}^{\Gamma}$. Then,

$$\mathsf{C}(P) \vdash \cdot; \Gamma,$$

with C as in Def. 3.2.

Proof. By induction on the structure of P.

(Case $P \in \mathsf{REQUESTS}^{\Gamma}$) The thesis follows from Lemma 3.5 on page 47.

(Case $P = (\nu a)(*a(s)[Q, R] \parallel S)$) We assume that P is part of a well-typed program, so we can take α, β such that $Q \triangleright \overline{\alpha}\{[\overline{\beta}]\}$ and $R \triangleright \overline{\beta}$. Moreover, by Def. 3.1, we know that $S \in \mathsf{SERVICES}^{\mathcal{G} \cup (a)}$. The induction hypothesis is that $\mathsf{C}(S) \vdash \cdot; \mathcal{G}, a : \mathcal{T}(\alpha)?\mathcal{T}(\beta)$.

By Def. 3.1, $Q \in \mathsf{SESSION}_{\checkmark}^{\emptyset,(s)}$ and $R \in \mathsf{SESSION}_{\times}^{\emptyset,(s)}$. Then, by Lemma 3.3 on the following page, $\mathfrak{serv}(Q\{s_1/s\}) \vdash s_1 : \overline{?\mathcal{T}(\alpha)}$, and by Lemma 3.4, $\mathfrak{exc}_{\perp}(S\{s_1'/s\}) \vdash s_1' : \mathcal{T}_{\perp}(\overline{\beta})$. Using $(\mathrm{T}\mathtt{some}_{\emptyset}^{s'})$ and $(\mathrm{T}\otimes)$, followed by an application of $(\mathrm{T}\mathsf{cut}^{?})$, the persistent channel a is of proper type and gets removed from the persistent context. □

Lemma 3.3

Take some $P \in \mathsf{SESSION}^{\emptyset,(s)}_{\checkmark}$. If $P \triangleright s : \overline{\alpha}\{[\overline{\beta}]\}$, then

$$\mathfrak{serv}(P) \vdash s : \overline{?\mathcal{T}(\alpha)},$$

with $\mathfrak{serv}$ as in Def. 3.4.

Proof. By induction on the structure of P.

(Case $P = \mathbf{throw}$) By the typing rules of CYH, we know that $\overline{\alpha}$ can be any type, so α can be anything. We need to show

$$\mathfrak{serv}(\mathbf{throw}) \vdash s : \overline{?\mathcal{T}(\alpha)} = \oplus(\oplus A \oplus \bot),$$

for appropriate A. This follows from an application of $(\mathrm{T}\oplus^{s}_{\emptyset})$, $(\mathrm{T}\oplus_2)$, and then $(\mathrm{T}\bot)$.

(Case $P = \mathbf{0}$) By the typing rules of CYH, we know $\overline{\alpha} = \mathtt{end}$, so $\alpha = \mathtt{end}$. We need to show

$$\mathfrak{serv}(\mathbf{0}) \vdash s : \overline{?\mathbf{1}} = \oplus(\oplus\bot \oplus \bot),$$

which follows from an application of $(\mathrm{T}\oplus^{s}_{\emptyset})$, $(\mathrm{T}\oplus_1)$, $(\mathrm{T}\oplus^{s_2}_{\emptyset})$, and then $(\mathrm{T}\bot)$.

(Case $P = s?(e).P'$) We know $\overline{\alpha} = \downarrow(\theta).\overline{\alpha'}$, so $\alpha = \uparrow(\theta).\alpha'$. We need to show

$$\mathfrak{serv}(s?(e).P') \vdash s : \overline{?(\theta \otimes \mathcal{T}(\alpha'))} = \oplus(\oplus(\theta \parr \overline{?\mathcal{T}(\alpha')}) \oplus \bot),$$

which follows from an application of $(\mathrm{T}\oplus^{s}_{\emptyset})$, $(\mathrm{T}\oplus_1)$, $(\mathrm{T}\oplus^{s_2}_{\emptyset})$, $(\mathrm{T}\parr)$, consuming the received channel with $C_e \vdash e : \theta$, and then the induction hypothesis. The output/branch/select cases can be shown in an analogous way, so we omit their proofs. □

Lemma 3.4

Take some $\mathcal{D} \in \mathcal{P}(\mathcal{C})$, some $b \in \{\mathbf{1}, \bot\}$, and some $P \in \mathsf{SESSION}^{\emptyset,\mathcal{D}}_{\times}$. If $P \triangleright x : \beta_x$ for every $x \in \mathcal{D}$, then

$$\mathfrak{exc}_b(P) \vdash (x : \mathcal{T}_b(\beta_x))_{x \in \mathcal{D}},$$

with $\mathfrak{exc}$ as in Def. 3.3.

Proof. This proof by induction on the structure of P follows directly from the converted types, and the induction hypothesis. □

Lemma 3.5

Take some $\mathcal{G} \in \mathcal{P}(\mathcal{C})$, and some $P \in \mathsf{REQUESTS}^{\mathcal{G}}$. Then,

$$\mathsf{C}(P) \vdash \cdot; \mathcal{G},$$

with C as in Def. 3.2.

Proof. By induction on the structure of P.

(Case $P = \overline{a}(s)[(s), Q, R]$) We assume that P is part of a well-typed program, so we can take α, β such that $Q \triangleright \alpha\{\![\beta]\!\}$ and $R \triangleright \beta$. Moreover, we can assume that the service channel $a \in \mathcal{G}$ is of proper type: $\mathcal{T}(\alpha)?\mathcal{T}(\beta)$. We need to show that

$$\mathsf{C}(\overline{a}(s)[(s), Q, R]) = \mathfrak{refine}_0^a(\mathcal{G}, \emptyset, \cdot, s, Q, R) \vdash \cdot; \underbrace{\mathcal{G}', a : \mathcal{T}(\alpha)?\mathcal{T}(\beta)}_{\mathcal{G}},$$

which follows from Lemma 3.6.

(Case $P = Q \parallel R$) Since $Q, R \in \mathsf{REQUESTS}^{\mathcal{G}}$, the thesis follows from the induction hypothesis and an application of (T$\parallel$). □

Lemma 3.6

Take some $\mathcal{G}, \mathcal{D} \in \mathcal{P}(\mathcal{C})$, some $\eta \in \mathcal{C} \cup \{\cdot\}$, some $s \notin \mathcal{D}$, some $P \in \mathsf{SESSION}_{\checkmark}^{\mathcal{G}, \mathcal{D} \cup (s)}$, and some $Q \in \mathsf{SESSION}_{\times}^{\emptyset, \mathcal{D} \cup (s)}$. If $P \triangleright (x : \alpha_x\{\![\beta_x]\!\})_{x \in \mathcal{D}}, s : \alpha^\star\{\![\beta^\star]\!\}$ and $Q \triangleright (x : \beta_x)_{x \in \mathcal{D}}, s : \beta^\star$, then

$$\mathfrak{refine}_0^a(\mathcal{G}, \mathcal{D}, \eta, s, P, Q) \vdash (x :?\mathcal{T}(\alpha_x))_{x \in \mathcal{D}}, \eta : \&\mathbf{1} \oplus (\bullet(\&\mathcal{T}(\beta_x))_{x \in \mathcal{D}}); \mathcal{G},$$

with $\mathfrak{refine}$ as in Def. 3.5.

Proof. The persistent context can be achieved by an appropriate amount of applications of (T$<$). The linear context remains to be proven.

We assume that the service channel $a \in \mathcal{G}$ is of proper type: $\mathcal{T}(\alpha^\star)?\mathcal{T}(\beta^\star)$. This means that, after an application of (Tcopy), we need to show that

$$\mathfrak{refine}_1^{\cdot}(\mathcal{G}, \mathcal{D}, \eta, s, P, Q) \vdash s : \mathcal{T}(\alpha^\star)?\mathcal{T}(\beta^\star).$$

W.l.o.g., assume $\eta = y \in \mathcal{C}$. Then, $\mathfrak{refine}_1^{\cdot} = \mathfrak{refine}_2^{\cdot}$. After an application of (T$\oplus_2$) on y, we have to show

$$\mathfrak{refine}_3^{\mathcal{D}}(\mathcal{G}, \mathcal{D}, y_1, s, P, Q) \vdash y_1 : \bullet(\&\mathcal{T}(\beta_x))_{x \in \mathcal{D}}.$$

Let $Q' := Q\{x'/x\}_{x \in \mathcal{D}}$, and $d := |\mathcal{D}|$. After an application of (T$\parr$) for every channel in $\mathcal{D}$, we need to show

$$\mathfrak{refine}_3^{\emptyset}(\mathcal{G}, \mathcal{D}, y_{d+1}, s, P, Q') \vdash y_{d+1} : \mathbf{1}, (x' : \&\mathcal{T}(\beta_x))_{x \in \mathcal{D}}.$$

We remove the channel y_{d+1} by an application of (T**1**), and have to show the typings for

$$\mathfrak{refine}_4^{\cdot}(\mathcal{G}, \mathcal{D}, \cdot, s, P, Q').$$

Let $\mathcal{D}' = \mathcal{D}\{x'/x\}_{x\in\mathcal{D}}$. First, we apply (T⅋) on s and create a new channel y with (Tcut). In the left process we apply (T&) on y. For the left case, we apply (T$\oplus^{y_1}_{\mathcal{D}'\cup(s')}$) and (T$\bot$). Several things need to be proven:

$$\mathfrak{refine}_5^{\mathcal{D}'\cup(s')}(\emptyset, \emptyset, \cdot, s', \mathbf{0}, Q'\{s'/s\}) \vdash (x' : \&\mathcal{T}(\beta_x))_{x\in\mathcal{D}}, s' : \&\mathcal{T}(\beta^\star), \tag{3.48}$$

$$\mathfrak{refine}_6^{\mathcal{D}'\cup(s')}(\emptyset, \emptyset, y_1, \cdot, \mathbf{0}, \mathbf{0}) \vdash y_1 : \overline{\bullet(\mathcal{B} \cup (\&\mathcal{T}(\beta^\star)))}, (x' : \&\mathcal{T}(\beta_x))_{x\in\mathcal{D}}, s' : \&\mathcal{T}(\beta^\star), \tag{3.49}$$

$$\mathfrak{req}(\mathcal{G}, \mathcal{D}\cup(s_1), \emptyset, \cdot, y, P\{s_1/s\}) \vdash (x :?\mathcal{T}(\alpha_x))_{x\in\mathcal{D}}, s_1 :?\mathcal{T}(\alpha^\star), y : \&\mathbf{1} \oplus (\bullet((\&\mathcal{T}(\beta_x))_{x\in\mathcal{D}}) \cup (\&\mathcal{T}(\beta^\star))). \tag{3.50}$$

To show (3.48), after an application of (T$\&^x_d$) to the channels in $\mathcal{D}'$ and to s', we need to show that

$$\mathfrak{refine}_5^{\emptyset}(\emptyset, \emptyset, \cdot, \cdot, \mathbf{0}, Q\{x'_1/x\}_{x\in(\mathcal{D}\cup(s))}) = \mathfrak{exc}(Q\{x'_1/x\}_{x\in(\mathcal{D}\cup(s))}) \vdash (x'_1 : \mathcal{T}(\beta_x))_{x\in\mathcal{D}}, s'_1 : \mathcal{T}(\beta^\star),$$

which follows from Lemma 3.4.

To show (3.49), we apply (T$\otimes$) for every channel in $\mathcal{D}'$ and for s', removing the channels from the content using (Tid). It remains to show that

$$\mathfrak{refine}_6^{\emptyset}(\emptyset, \emptyset, y_{d+2}, \cdot, \mathbf{0}, \mathbf{0}) \vdash y_{d+2} : \bot,$$

which we achieve by applying (T$\bot$).

(3.50) follows from Lemma 3.7 on the next page, or from its proof's induction hypothesis when it refers to this lemma. □

Lemma 3.7

Take some $\mathcal{G}, \mathcal{D} \in \mathcal{P}(\mathcal{C})$, some $\Lambda \in \mathcal{P}(\mathcal{C} \times \mathbb{N}_0)$, some $\mu, \eta \in \mathcal{C} \times \mathbb{N}_0 \cup \{\cdot\}$, and some $P \in \mathsf{SESSION}^{\mathcal{G},\mathcal{D}}_{\checkmark}$. If

i) $P \triangleright \alpha_x\{\!|\beta_x|\!\}$ for every $x \in \mathcal{D}$,
ii) $n \geq \mathfrak{card}(P)$ for every $(y, n) \in \Lambda$, and
iii) $\mu = (z, o)$ implies $o \geq \mathfrak{card}(P)$ and $o \geq n$ for every $(y, n) \in \Lambda$,

then

$$\begin{aligned}\mathfrak{req}(\mathcal{G}, \mathcal{D}, \Lambda, \mu, \eta, P) \vdash\ & (x :?\mathcal{T}(\alpha_x))_{x\in\mathcal{D}},\\ & (y : \|_n)_{(y,n)\in\Lambda}, (z : \overline{\|}_o)_{\text{if } \mu=(z,o)},\\ & \eta : \&\mathbf{1} \oplus (\bullet(\&\mathcal{T}(\beta_x))_{x\in\mathcal{D}}),\end{aligned}$$

with $\mathfrak{req}$ as in Def. 3.6.

Proof. By induction on the structure of P.

(Case $P = \mathbf{throw}$) We have to show the typings for

$$\mathfrak{throw}_0(\mathcal{D}, \Lambda, \mu, \eta),$$

which follows from Lemma 3.8 on page 52.

(Case $P = \mathbf{0}$) By the typing rules of CYH, we know $\mathbf{0} \triangleright (\alpha_x = \mathsf{end})_{x\in\mathcal{D}}$. At certain steps, we need to account for two cases, depending on the liveness of the parallelism channels. With $\mathfrak{getlive}$ as in Def. 3.13 and $\mathfrak{islive}$ as in Def. 3.14, let $\mathfrak{L} := ((\Lambda \neq \emptyset \implies \mathfrak{getlive}(\Lambda) = \emptyset)$ and not $\mathfrak{islive}(\mu))$. We have to show the typing for

$$\mathfrak{reqstep}_0(\mathcal{G}, \mathcal{D}, \Lambda, \mu, \eta, \cdot, \mathbf{0}),$$

with $\mathfrak{reqstep}$ as in Def. 3.7.

If $\mu = (z, 0)$, we apply $(\mathrm{T}\&^z_{\mathcal{D}\cup\mathfrak{chans}(\Lambda)})$, with $\mathfrak{chans}$ as in Def. 3.11, to obtain

$$\mathfrak{reqstep}^{\cdot}_1(\mathcal{G}, \mathcal{D}, \Lambda, (z_1, 0), \eta, \cdot, \mathbf{0}) \vdash z_1 : \oplus\bot \oplus \bot.$$

If $\mu = (z, o{+}1)$, we apply the same rule to obtain

$$\mathfrak{reqstep}^{\cdot}_1(\mathcal{G}, \mathcal{D}, \Lambda, (z_1, o{+}1), \eta, \cdot, \mathbf{0}) \vdash z_1 : \oplus\overline{\|}_o \oplus \bot.$$

If $\mathfrak{L}$, $\mathfrak{reqstep}^{\cdot}_1 = \mathfrak{reqstep}^{\mathcal{D}}_2$, so, on every channel $x \in \mathcal{D}$, we apply $(\mathrm{T}\&^x_d)$ and then $(\mathrm{T}\&)$. The right case follows from an application of $(\mathrm{T}\mathbf{1})$, and Lemma 3.8 on page 52 and Lemma 3.9 on page 53. For the left case, we continue with the next channel until we arrive at the next step. We have to show

$$\mathfrak{reqstep}^{\emptyset}_2(\mathcal{G}, \mathcal{D}^2, \Lambda, \mu^1, \eta, \cdot, \mathbf{0}) \vdash (x_2 : \&\mathbf{1})_{x\in\mathcal{D}}.$$

Otherwise, let $\mathcal{D}^2 := \mathcal{D}$. We have to show

$$\mathfrak{reqstep}^{\emptyset}_2(\mathcal{G}, \mathcal{D}^2, \Lambda, \mu^1, \eta, \cdot, \mathbf{0}) \vdash (x :?\mathbf{1})_{x\in\mathcal{D}}.$$

If $\mathfrak{L}$, $\mathfrak{reqstep}_2^{\emptyset} = \mathfrak{reqstep}_3^{\Lambda}$. We apply a similar process as above to the channels $(y,0) \in \Lambda$, and have to show

$$\mathfrak{reqstep}_3^{\emptyset}(\mathcal{G}, \mathcal{D}^2, \Lambda^2, \mu^1, \eta, \cdot, \mathbf{0}) \vdash (y_2 : \&\mathbf{1})_{(y,0)\in\Lambda}.$$

Otherwise, $\mathfrak{reqstep}_2^{\emptyset} = \mathfrak{reqstep}_3^{\mathfrak{live}(\Lambda)}$. We also apply that similar process as above, but only to those channels $(y, n{+}1) \in \mathfrak{live}(\Lambda)$. Let $\Lambda^2 := \Lambda\{y_2/y\}_{(y,n+1)\in\mathfrak{live}(\Lambda)}$. We have to show

$$\begin{aligned}\mathfrak{reqstep}_3^{\emptyset}(\mathcal{G}, \mathcal{D}^2, \Lambda^2, \mu^1, \eta, \cdot, \mathbf{0}) \vdash\ & (y_2 : \&\|_n)_{(y,n+1)\in\mathfrak{live}(\Lambda)},\\ & (y : \|_0)_{(y,0)\in\Lambda\setminus\mathfrak{live}(\Lambda)}.\end{aligned}$$

If $\mu = (z, 0)$, we apply and (T$\oplus_1$) and (T$\oplus^{z_2}_{\mathcal{D}^2\cup\mathfrak{chans}(\Lambda^2)}$) to obtain

$$\mathfrak{reqstep}_4^{\cdot}(\mathcal{G}, \mathcal{D}^2, \Lambda^2, (z_3, 0), \eta, \cdot, \mathbf{0}) \vdash z_3 : \bot.$$

If $\mu = (z, o{+}1)$, we apply the same rules to obtain

$$\mathfrak{reqstep}_4^{\cdot}(\mathcal{G}, \mathcal{D}^2, \Lambda^2, (z_3, o{+}1), \eta, \cdot, \mathbf{0}) \vdash z_3 : \overline{\|_o}.$$

If $\mathfrak{L}$, we can finish the proof. We apply (T$\&^x_d$) and (T$\mathbf{1}$) to every channel in $\mathcal{D}$ and Λ. If $\mu = (z, 0)$ (it will not be $(z, o{+}1)$ since not $\mathfrak{islive}(\mu)$), we apply (T$\bot$) to z_3. Finally, we apply (T$\oplus_1$) to η and then (T$\&^{\eta^1}$).

Otherwise, $\mathfrak{reqstep}_4^{\cdot} = \mathfrak{reqstep}_5^{\mathfrak{live}(\Lambda)}$. We apply (T$\&^x_d$) to every channel in $\mathfrak{live}(\Lambda)$. Let $\Lambda^3 := \Lambda\{y_3/y\}_{(y,n+1)\in\mathfrak{live}(\Lambda)}$. We obtain

$$\begin{aligned}&\mathfrak{reqstep}_5^{\emptyset}(\mathcal{G}, \mathcal{D}^2, \Lambda^3, \mu^3, \eta, \cdot, \mathbf{0})\\ =\ &\mathfrak{reqstep}_0^{\cdot}(\mathcal{G}, \mathcal{D}^2, \mathfrak{nextstep}(\Lambda^3), \mathfrak{nextstepmaster}(\mu^3), \eta, \cdot, \mathbf{0})\\ &\vdash (x :?\mathbf{1})_{x\in\mathcal{D}},\\ &\quad (y_3 : \|_n)_{(y,n+1)\in\mathfrak{live}(\Lambda)}, (y : \|_0)_{(y,0)\in\Lambda},\\ &\quad (z_3 : \overline{\|_o})_{\text{if } \mu=(z,o+1)},\\ &\quad \eta : \&\mathbf{1} \oplus (\bullet(\&\mathcal{T}(\beta_x))_{x\in\mathcal{D}}).\end{aligned}$$

Since the applications of $\mathfrak{nextstep}$ (Def. 3.17) and $\mathfrak{nextstepmaster}$ (Def. 3.18) guarantee that at some point $\mathfrak{L}$ will be true, we can show this by the induction hypothesis.

(Case $P = d?(e).P'$) We know $\alpha_d = \downarrow(\theta).\alpha'_d$, where $P' \triangleright d : \alpha'_d\{[\![\beta_d]\!]\}, (x : \alpha_x\{[\![\beta_x]\!]\})_{x\in\mathcal{D}\setminus(d)}$. Since $\mathfrak{card}(P) \geq 1$, we have $\Lambda \neq \emptyset \implies \mathfrak{live}(\Lambda) = \Lambda$ and $\mu = (z, o) \implies \mathfrak{islive}(\mu)$. We have to show the typings for

$$\mathfrak{reqstep}_0^{\cdot}(\mathcal{G}, \mathcal{D}, \Lambda, \mu, \eta, d, d?(e).P').$$

If $\mu = (z, o{+}1)$, we apply (T$\&^z_{\mathcal{D}\cup\mathfrak{chans}(\Lambda)}$) to obtain

$$\mathfrak{reqstep}_1^{\cdot}(\mathcal{G}, \mathcal{D}, \Lambda, (z_1, o{+}1), \eta, d, d?(e).P') \vdash z_1 : \oplus\overline{\|_o} \oplus \bot.$$

Otherwise, $\mathfrak{reqstep}_0^{\cdot} = \mathfrak{reqstep}_1^{\cdot}$.

Since $\mathfrak{reqstep}_1^{\cdot} = \mathfrak{reqstep}_2^{(d)}$, we apply (T&$_d^x$) and (T&). The right case follows from an application of (**T1**), and Lemma 3.8 on the next page and Lemma 3.9 on page 53. For the left case, we have to show

$$\begin{aligned}\mathfrak{reqstep}_2^{\emptyset} = \mathfrak{reqstep}_3^{\mathfrak{live}(\Lambda)}(\mathcal{G}, \mathcal{D}\{d_2/d\}, \Lambda, \mu^1, \eta, d_2, d_2?(e).P'\{d_2/d\})\\ \vdash d_2 : \&(\theta\mathfrak{Y}?\mathcal{T}(\alpha'_d)).\end{aligned}$$

Since $\mathfrak{live}(\Lambda) = \Lambda$, we apply the same process to the channels in Λ as we did to d at $\mathfrak{reqstep}_2^{(d)}$, obtaining

$$\mathfrak{reqstep}_3^{\emptyset}(\mathcal{G}, \mathcal{D}\{d_2/d\}, \Lambda^2, \mu^1, \eta, d, d_2?(e).P'\{d_2/d\}) \vdash (y_2 : \&\|_n)_{(y,n+1)\in\Lambda}.$$

If $\mu = (z, o{+}1)$, we apply (T&$_1$) to z_1 and then (T&$^{z_2}_{\mathcal{D}\{d_2/d\}\cup\Lambda^2}$) to obtain

$$\mathfrak{reqstep}_4^{\cdot}(\mathcal{G}, \mathcal{D}\{d_2/d\}, \Lambda^2, (z_3, o{+}1), \eta, d, d_2?(e).P'\{d_2/d\} \vdash z_3 : \overline{\|_o}.$$

Otherwise, $\mathfrak{reqstep}_3^{\emptyset} = \mathfrak{reqstep}_4^{\cdot}$.

Now, we apply (T&$_d^{d_2}$) and then (T&$_d^{y_2}$) to every $(y, n{+}1) \in \Lambda$ to obtain

$$\mathfrak{reqstep}_5^{\emptyset}(\mathcal{G}, \mathcal{D}\{d_3/d\}, \Lambda^3, \mu^3, \eta, d_3, d_3?(e).P'\{d_3/d\}) \vdash d_3 : \theta\mathfrak{Y}?\mathcal{T}(\alpha'_d), (y_3 : \|_n)_{(y,n+1)\in\Lambda}.$$

Finally, we apply (T$\mathfrak{Y}$) to d_3, consuming the received channel with $C_e \vdash e : \theta$. It remains to show

$$\mathfrak{req}(\mathcal{G}, \mathcal{D}\{d_4/d\}, \mathfrak{nextstep}(\Lambda^3), \mathfrak{nextstepmaster}(\mu^3), \eta, P'\{d_4/d\}) \vdash d_4 :?\mathcal{T}(\alpha'_d),$$

which follows from the induction hypothesis. The output/branch/select cases can be shown in an analogous way, so we omit their proofs.

(Case $P = P_1 \parallel P_2$) By the grammar in Def. 3.1, we know there are $\mathcal{D}_1, \mathcal{D}_2 \in \mathcal{P}(\mathcal{C})$ such that $\mathcal{D}_1 \cap \mathcal{D}_2 = \emptyset$ and $\mathcal{D}_1 \cup \mathcal{D}_2 = \mathcal{D}$, and that

$$P_1 \triangleright (x : \alpha_x\{\![\beta_x]\!\})_{x\in\mathcal{D}_1}, \text{ and } P_2 \triangleright (x : \alpha_x\{\![\beta_x]\!\})_{x\in\mathcal{D}_2}.$$

Let $\ell := \mathfrak{card}(P_1 \parallel P_2)$. To justify the cut on the newly created parallelism channel, we have to show

$$\begin{aligned}\mathfrak{req}(\emptyset, \mathcal{D}_1, \Lambda \cup (y, \ell), \mu, \eta, P_1) \vdash\ & (x :?\mathcal{T}(\alpha_x))_{x\in\mathcal{D}_1},\\ & (y : \|_n)_{(y,n)\in\Lambda\cup(y,\ell)},\\ & (z : \overline{\|_o})_{\text{if } \mu=(z,o)},\\ & \eta : \&\mathbf{1} \oplus (\bullet(\&\mathcal{T}(\beta_x))_{x\in\mathcal{D}}),\end{aligned}$$

and

$$\begin{aligned}\mathfrak{req}(\emptyset, \mathcal{D}_2, \emptyset, (z, \ell), \cdot, P_2) \vdash\ & (x :?\mathcal{T}(\alpha_x))_{x\in\mathcal{D}_2},\\ & z : \overline{\|_\ell}.\end{aligned}$$

Both follow from the induction hypothesis.

(Case $P = \overline{a}(s)[\mathcal{D} \cup (s), Q, R]$) By the grammar in Def. 3.1, we can assume $\Lambda = \emptyset$, $\mu = \cdot$, and $\eta = y$. We have to show

$$\mathfrak{refine}_0^a(\mathcal{G}, \mathcal{D}, y, s, Q, R) \vdash (x : ?\mathcal{T}(\alpha_x))_{x \in \mathcal{D}}, y : \&\mathbf{1} \oplus (\bullet(\&\mathcal{T}(\beta_x))_{x \in \mathcal{D}}),$$

which follows from Lemma 3.6. □

Lemma 3.8

Take some $\Delta \in \mathcal{P}(\mathcal{C})$, some $\Lambda \in \mathcal{P}(\mathcal{C} \times \mathbb{N}_0)$, some $\mu \in \mathcal{C} \times \mathbb{N}_0 \cup \{\cdot\}$, and some $\eta \in \mathcal{C} \cup \{\cdot\}$. Then,

$$\begin{aligned} \mathfrak{throw}_0(\Delta, \Lambda, \mu, \eta) \vdash\ & (x : \&A_x)_{x \in \Delta}, \\ & (y : \&A_y)_{(y,n) \in \Lambda}, \\ & (z : \oplus(A_z \oplus \bot))_{\text{if } \mu = (z,o)}, \\ & \eta : \&\mathbf{1} \oplus A_\eta, \end{aligned}$$

with every $A_\circ$ arbitrary and $\mathfrak{throw}$ as in Def. 3.9.

Proof. We have to show the typings for Δ, Λ, and η in

$$\mathfrak{throw}_1(\Delta, \Lambda, \cdot, \eta), \tag{3.51}$$

and, if $\mu = (z, o)$, the typing for z in

$$\mathfrak{throw}_3(\emptyset, \emptyset, (z, o), \cdot). \tag{3.52}$$

To show (3.51), we have to show the typings for Δ and Λ in

$$\mathfrak{throw}_2(\Delta, \Lambda, \cdot, \cdot),$$

which is achieved by an application of (T&^{x}) for each of those channels. Moreover, if $\eta = x$, we have to show its typing in

$$\mathfrak{throw}_4(\emptyset, \emptyset, \cdot, x),$$

which is achieved by applying (T&$_1$), then (T&$_d^x$), and finally (T$\mathbf{1}$).

To show (3.52), we apply (T&$_\emptyset^z$). Now, we need to show

$$\mathfrak{throwmaster}(z_1) \vdash A_z \oplus \bot,$$

which follows from Lemma 3.9 on the next page. □

Lemma 3.9

Take some $\mu \in \mathcal{C} \cup \{\cdot\}$. Then,

$$\mathfrak{throwmaster}(\mu) \vdash \mu : A \oplus \bot,$$

with A arbitrary and $\mathfrak{throwmaster}$ as in Def. 3.10.

Proof. If $\mu = \cdot$, the thesis is vacuously true. Otherwise, it follows from an application of ($\mathrm{T}\oplus_2$) followed by ($\mathrm{T}\bot$). □

3.4.2 Deadlock-freeness

Our well-typedness result has the desired corollary that the resulting processes from our conversion are deadlock-free. In combination with operational correspondence results (cf. Section 3.4.3), this leads to a much simpler deadlock-freeness proof for the fragment $\mathsf{CYH}^{\mathsf{CH}}$ than the one given in [10].

Our definition of deadlock-freeness is an extension of the Progress Theorem from [4, Thm. 3.2, p. 15]. It does not only guarantee for live processes (processes with a communication prefix that is not a request or replication), but it also guarantees that pending service requests can be answered at any moment. Before we give the definition, we need to formally define liveness, for which we also need a non-prefixed (static) context.

Definition 3.24

The static context C is defined as follows.

$$C ::= C \parallel C \mid C \oplus C \mid (\nu s)C \mid P \mid -$$

Definition 3.25 (Liveness)

A process $P \in \mathsf{DCPTP}$ is $\mathfrak{live}$ (notation $\mathfrak{live}(P)$) if and only if $P \equiv C[\pi.P']$, where π is a communication prefix (i.e. input, output, branch, or select).

Definition 3.26 (Deadlock-freeness)

A process $P \in \mathsf{DCPTP}$ is deadlock-free if

i) $P \vdash \cdot; \Gamma$,
ii) $P \equiv C[(\nu s)(\overline{a}\langle s\rangle \parallel Q)]$ implies $P \equiv C'[(\nu a)(\,!a(s); R \parallel R')]$, $P \to S \equiv C[(\nu s)(R \parallel Q)]$, and S is deadlock-free, and
iii) $\mathfrak{live}(P)$ implies there is $T \in \mathsf{DCPTP}$ such that $P \to T$ and T is deadlock-free.

Corollary 3.10

Given some well-typed program $P \in \mathsf{CYH}^{\mathsf{CH}}$, the converted process $\mathsf{C}(P)$ is deadlock-free, with C as in Def. 3.2.

Proof. The proof relies on the main results from [4], which transfer to our system since we use (structurally) identical inference and reduction rules.

By Theorem 3.1, $\mathsf{C}(P) \vdash \cdot;\cdot$, satisfying condition (i).

Assume $\mathsf{C}(P) \equiv C[(\nu s)(\overline{a}\langle s\rangle \parallel Q)]$. This can only happen if P contains some request to a service on channel a. By our well-typedness assumption, P contains the corresponding service definition, so $\mathsf{C}(P) \equiv C'[(\nu a)(\,!a(s); R \parallel R')]$. By Lemmas 3.3 and 3.7, Q and R are of dual type, so $\mathsf{C}(P) \to S \equiv C[(\nu s)(R \parallel Q)]$. By the Type Preservation Theorem [4, Thm. 3.2, p. 15], $S \vdash \cdot;\cdot$, so we can show that S is deadlock-free by an easy induction proof. This satisfies condition (ii).

Assume $\mathfrak{live}(\mathsf{C}(P))$. By the Progress Theorem [4, Thm. 3.3, p. 15], there is a T such that $\mathsf{C}(P) \to T$. By another application of the Type Preservation Theorem, $T \vdash \cdot;\cdot$. Again, a simple proof by induction will show that T is deadlock-free. This shows condition (iii). □

3.4.3 Discussion: operational correspondence

According to Gorla, one criterion for a valid encoding (we use the word "conversion" in this thesis) is *operational correspondence* [25]. It consists of two parts: *completeness*, the notion that a reduction in the source process can be simulated in the conversion, and *soundness*, the notion that a reduction in a converted process reflects a reduction in the source process. We conjecture that our conversion has (a form of) both completeness and soundness.

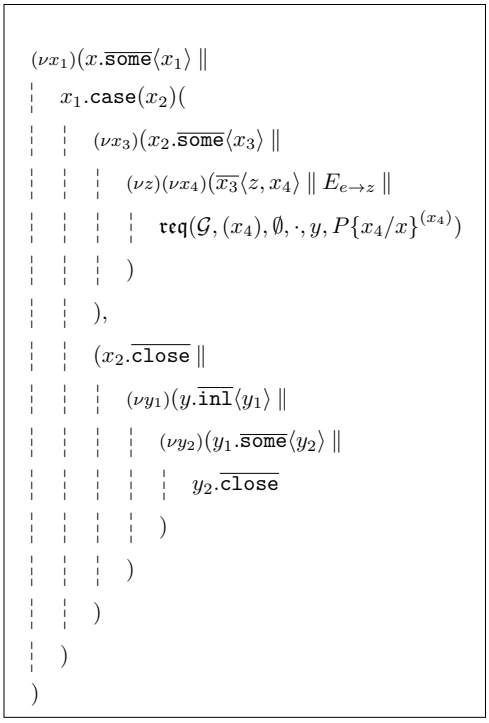


Figure 3.2: Conversion of the default handler in equation (3.53)

In [10] a liveness proof for CYH is given, which is a condition on processes that is equivalent to our definition of deadlock-freeness (see Def. 3.26). The proof is, however, not trivial, whereas our deadlock-freeness result is in fact a mere result of the Curry-Howard correspondence with linear logic. An operational correspondence proof would allow us to transfer our deadlock-freeness result to the source fragment $\mathsf{CYH}^{\mathsf{CH}}$.

Unfortunately, due to the synchronization steps in our conversion, formalizing our completeness and soundness conjectures is not straightforward. Consider the following $\mathsf{CYH}^{\mathsf{CH}}$ request:

$$\overline{a}(x)[(x), x!\langle e\rangle.P^{(x)}, Q^{(x)}] \tag{3.53}$$

The conversion of the default handler is given in Figure 3.2, where y is the channel over which the exception handler can be invoked. Let R denote the process in this figure. When no exceptions are thrown, it takes three steps before the actual send action occurs, so after four steps, one step from the original process has been taken:

$$R \to^4 \mathfrak{req}(\mathcal{G}, (x_4), \emptyset, \cdot, y, P\{x_4/x\}^{(x_4)})$$

Another difficulty is that parallel communications always happen synchronously in converted processes.

To overcome these problems, we need to find an alternative definition of reduction for which we can prove completeness and soundness results. For the time being, we informally use $\mapsto$ to denote a step in $\mathsf{CYH}^{\mathsf{CH}}$ in which parallel communications happen synchronously, and $\rightarrowtail$ to denote a step of actual communication in converted processes. An intuition towards defining such steps is to use the types of the channels. For example, the type of x in Fig. 3.2 is $?\mathcal{T}(\uparrow(\theta).\alpha') = \&(\&(\theta \otimes ?\mathcal{T}(\alpha')) \,\&\, \mathbf{1})$. One $\rightarrowtail$-step would involve reductions up to the point that the type of x is $?\mathcal{T}(\alpha')$.

There are more issues. Once services and requests connect, the semantics of CYH turn the processes into runtime processes, which uses protection constructs for exception handling and buffers for asynchronous communication. Our conversion is, however, not defined on such runtime processes, so we cannot directly relate them. Similar issues arise when exceptions are thrown and exception handlers are invoked. Communication in exception handlers does not need to be synchronised, so this needs to be taken into account.

The buffered semantics from Section 2.2 could come in handy here. We can use the equalities in Definition 2.6 to convert asynchronous process terms to buffered terms, denoted $\square P$. We would then also need to define a notion of equivalence between buffered DCPTP processes and runtime CYH processes. Without formally defining it, we denote this relation using $- \mathrel{\tilde{\square}} -$.

We believe that it is possible to prove an operational correspondence between source processes and our conversion of them using the methods above. We are ready to state our conjectures. $\rightarrow^*$ denotes the transitive, reflexive closure of $\rightarrow$.

Conjecture 3.11 (Completeness)

For every $P \in \mathsf{CYH}^{\mathsf{CH}}$ and runtime CYH process Q such that $P \mapsto Q$, there exists $R \in \mathsf{DCPTP}$ such that $\mathsf{C}(P) \rightarrow^* R$ and $\square R \mathrel{\tilde{\square}} Q$.

Conjecture 3.12 (Soundness)

For every $P \in \mathsf{CYH}^{\mathsf{CH}}$ and $R \in \mathsf{DCPTP}$ such that $\mathsf{C}(P) \rightarrowtail R$, there exists runtime CYH process Q such that $P \rightarrow^* Q$ and $\square R \mathrel{\tilde{\square}} Q$.

We find these statements plausible, because our conversion was developed by carefully examining the way in which CYH processes reduce. The fact that we can not state a direct operational correspondence, but have to use intermediate correspondences, demonstrates the limitations of using a Curry-Howard foundation to implement fault-tolerance through exception handling in message-passing concurrency.

Chapter 4

Future and related work

In this chapter we discuss several directions for further research on fault-tolerance in Curry-Howard session type theory. We discuss a way of exploring fault-tolerance in multiparty session types, which are more expressive than the binary ones we have discussed so far. Moreover, we discuss the validity of our conversion, and mention some fault-tolerant session calculi besides the one from Carbone et al. [10, 11], and discuss how they might relate to the research in this thesis. Finally, we discuss some extensions to session types that could be incorporated into our binary calculus.

4.1 Multiparty session types

Multiparty session types allow you to design communication protocols between multiple parties, from a global perspective. It is then possible to obtain local perspectives of the protocol for each participant by means of projections. Multiparty session types were introduced by Honda et al. [30].

The multiparty calculus can be more user-friendly than the binary one for designing protocols. Consider for example a protocol with three participants. In the binary systems, they need to be connected in series, i.e. the first communicates with the second, and the second with the first and third, but the third not (directly) with the first. If the first and third participants want to communicate, the second has to forward their messages to each other. This clutters their session types, burying the intended protocol in housekeeping.

Despite the expressive advantages of the multiparty calculus, its global perspective makes it more complex to prove results about. For this reason, Caires and Pérez presented a way of analysing multiparty protocols in a synchronous binary calculus [5], based on intuitionistic linear logic [22, 23, 3].

In Appendix D, we present such a multiparty session calculus based on the one by Honda [30] and the slightly more expressive version by Caires and Pérez [5]. We show how it relates to our binary calculus, by exhibiting a conversion from multiparty to binary in the style of Caires and Pérez [5]. The appendix also contains some results about the multiparty system and our conversion.

The conversion shows that multiparty session types are synchrony-agnostic, since our binary system is asynchronous, and the relation has previously only been shown for synchronous binary session types. Another novelty is that the classical basis, as opposed to the intuitionistic basis in the original conversion, of our binary system has no (significant) influence on the relation.

We also give a multiparty version of the monadic session type for non-determinism, which relates to non-determinism in the binary system naturally. One direction of future research would be to explore how this newly introduced modality can be used to implement fault-tolerance. An important challenge is to determine whether it would be possible to implement exception handlers using such a type.

Another direction is to study the multiparty session type system by Capecchi et al. [9], which supports fault-tolerance in the form of global try-catch blocks, but has no logical basis in the form of a Curry-Howard correspondence. It would be interesting to explore the possibilities of projecting such global types into our binary type system from Chapter 2, in a way similar to our conversion in Chapter 3. Because the work by Capecchi et al. represents the multiparty extension of the system by Carbone et al. [10, 11] in many ways, it would be an intriguing approach to see whether it is possible to project the global types into the fragment of the system from Carbone et al. we support in our conversion. Because we can convert this fragment into our binary system, this would simplify the relation, and might give us deadlock-freeness for free. However, such a conversion would be limited, because the system permits nested try-catch blocks, even in exception handlers, whereas this is not allowed in CYH.

A similar multiparty session type system is presented by Adameit et al. in [1]. Instead of using global try-catch blocks, they extend multiparty sessions with optional blocks. An optional block is a prefix to another global type. This prefixed global type gets executed when the optional block is successfully finished, or when failure occurs anywhere inside it. This essentially means that the prefixed global session is actually the optional block's exception handler, and additional optional blocks would then be allowed in exception handlers.

4.2 Validity of the conversion

In an effort to standardize the comparison of process calculi, Gorla has proposed five criteria for an encoding (or conversion, as we call it in this thesis) to be valid [25]. We summarize them briefly:

- Compositionality: the encoding of subterms is independent from the context they appear in.
- Name invariance: the encoding does not depend on the specific names of terms, i.e. there is no difference between name substitution before or after encoding.
- Operational correspondence: reductions in the source process can be simulated in the encoding (completeness), and reductions in the encoding are reflected by reduction in the source process (soundness).
- Divergence reflection: the encoding does not *add* infinite behaviour, i.e. all infinite behaviour in the encoding originates from infinite behaviour in the source process.
- Success sensitiveness: given some notion of successful computation in the source and target calculi, when a source process may reach success, the encoding will as well.

We already discussed operational correspondence in Section 3.4.3, although the conjectures in this section remain to be proven. Compositionality is a plausible property of our conversion, albeit up to a limited notion of independence, due to the way parallel communications have to synchronize. Name invariance has to be investigated, which might be a challenge due to the fact that our asynchronous calculus constantly requires new channel names. Divergence reflection is a free property of our conversion, since we do not allow processes with infinite behaviour in our fragment $\mathsf{CYH}^{\mathsf{CH}}$. Usually, success is expressed by an output on some specially reserved free name [37, 26]. Success sensitiveness may be provable, but we would need to add such a notion of success to both $\mathsf{CYH}^{\mathsf{CH}}$ and to its conversion.

4.3 Related work

After the work by Carbone et al. [10, 11], Mostrous and Vasconcelos presented another binary session type system with fault-tolerance in [36]. Their system focusses on *affine types*, which can be removed from the context of a process without explicitly consuming it. This allows them to add a cancellation primitive to their process algebra, which resembles the throwing of exceptions, but not the handling of them.

Fowler et al. presented a fault-tolerant, session typed, functional programming language [19], drawing from the work by Mostrous and Vasconcelos. It is different from the exceptional functional language by Caires and Pérez [4] in that it does not rely on an underlying logical system. Another difference is that it is asynchronous, which we have shown to not be an issue by redefining the system by Caires and Pérez to be asynchronous (cf. Chapter 2). It would be interesting to compare the expressivity of the work by Fowler et al., and that of the fragment $\mathsf{CYH}^{\mathsf{CH}}$, which we can convert to our logical system (cf. Chapter 3).

An early connection between session types and exception handling is presented by Dezani-Ciancaglini et al. in [18]. They use session types to define an object oriented programming language. When a property of an object is accessed and it is undefined, a *null pointer exception* is thrown. When that happens, the entire program gets cancelled, i.e. there is no exception handling. It would be interesting to apply the state-of-the-art of fault-tolerant exception types to add exception handling to this programming language.

Another early connection can by found in [32] by Hu et al. They present an extension of the Java programming language based on session type theory. They support try-catch blocks to handle exceptions in IO and session connection. However, the use of session type theory and fault-tolerance seem to be disjoint: the session principals stem from the theory, but exception handling was already a part of Java. It might be interesting to connect this application of session types with the theory more formally, using the now existing theory on fault-tolerance in session types.

The final paper including fault-tolerance we mention is [40] by Vieira et al. They present a session calculus similar to the π-calculus, which they call the Conversion Calculus (CC). This system supports try-catch blocks in a way that is very similar to that of Carbone et al. [10, 11]. However, the work does not include a type discipline. In a follow-up work, Caires and Vieira do present a type system for CC [8]. However, the version of CC in this paper does not include the aforementioned try-catch blocks. It would be interesting to see whether they can be included in the type system for CC.

4.4 Extensions to the binary calculus

As we mention in "Limitation 3: Recursion" of our fragment of CYH given in Section 3.2, our binary calculus has no support for recursion. In [39], Toninho et al. present a Curry-Howard session type system, extended with support for infinite behaviour using corecursion. In order to support the recursion of CYH, their method could be adapted to our type system. Just as our system does (cf. Section 3.4), their system has a Type Preservation and a Progress Theorem [39, Thm. 1 and 2, p. 10]. It is plausible that these results remain when we extend our system with corecursion, making it possible to add recursion to the fragment of CYH we support in our conversion.

For our conversion, we have focussed on the version of CYH presented in [11]. In the original presentation [10], however, there is support for *session delegation*. With session delegation, it is not only possible to send values of primitive types, but also channels of session types. Support for session delegation was already present in the original paper on the Curry-Howard correspondence for concurrency [7], and so is it in our type system. However, by using abstract generator and consumer processes (cf. the list of conventions from Section 3.3) in our conversion, we have abstracted away from dealing with the values that are being sent. To allow session delegation in our conversion, it is necessary to undo this abstraction. This would also make it possible to support conditional statements (cf. "Limitation 4: Conditional statements" from Section 3.2).

Chapter 5

Conclusion

We have successfully exhibited a way of modelling fault-tolerance through exception handling in a process calculus that is supported by strong logical foundations by means of a Curry-Howard correspondence with classical linear logic. We have done so by identifying a fragment of a process calculus with exception handling without such a logical foundation, (cf. Section 3.2), and by defining a conversion from that fragment to our logical calculus (starting in Def. 3.2).

The fact that this conversion only supports a fragment of the source calculus illustrates how complex the typed calculus of Carbone et al. is. This limited support is, however, not only due to the complexity of the source calculus. The modalities for non-deterministically available behaviour we have used can be seen as a unary version of branching: instead of a choice between multiple options, there is a choice between behaviour or no behaviour. Indeed, our conversion demonstrates that this is conceptually different from exception handling.

The result of the conversion does, however, give a clear means of understanding the intricacies of and logic behind exception handling. Converted processes are deadlock-free (cf. Corol. 3.10), and there is strong evidence for an operational correspondence between the source calculus and its conversion (cf. Section 3.4.3). This would significantly simplify the proof of deadlock-freeness in the source process calculus, since our results may transfer to it through our conversion.

We have also shown that the method of transforming a synchronous π-calculus into an asynchronous one is straightforward; it was a routine job to make the process calculus from [4] asynchronous. The classical logical basis did not induce a significant difference to an intuitionistic one. Also, adapting the modalities for non-determinism did not give any problems. However, using the asynchronous calculus can be tedious, due to the constant need for new names. This makes processes difficult to read, and has had an effect on the complexity of our conversion (see for example Def. 3.7, in which names change in almost every consecutive step of the conversion).

This thesis has laid a solid foundation to further research the logic behind exception handling. Its place in multiparty session type theory is yet to be researched. Moreover, its influence on extensions to session type theory, such as recursion, is an interesting subject for further studies.

References

[1] Manuel Adameit, Kirstin Peters, and Uwe Nestmann. Session types for link failures. In Ahmed Bouajjani and Alexandra Silva, editors, *Formal Techniques for Distributed Objects, Components, and Systems*, Lecture Notes in Computer Science, pages 1–16. Springer International Publishing, 2017. ISBN: 978-3-319-60225-7.

[2] Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3):297–347, June 1, 1992. ISSN: 0955-792X. DOI: 10.1093/logcom/2.3.297.

[3] Andrew Barber. Dual Intuitionistic Linear Logic. Technical Report ECS-LFCS-96-347, University of Edinburgh, 1996.

[4] Luís Caires and Jorge A. Pérez. Linearity, control effects, and behavioral types. In Hongseok Yang, editor, *Programming Languages and Systems*, Lecture Notes in Computer Science, pages 229–259. Springer Berlin Heidelberg, 2017. ISBN: 978-3-662-54434-1.

[5] Luís Caires and Jorge A. Pérez. Multiparty session types within a canonical binary theory, and beyond. In Elvira Albert and Ivan Lanese, editors, *Formal Techniques for Distributed Objects, Components, and Systems*, Lecture Notes in Computer Science, pages 74–95. Springer International Publishing, 2016. ISBN: 978-3-319-39570-8.

[6] Luís Caires, Jorge A. Pérez, Frank Pfenning, and Bernardo Toninho. Behavioral polymorphism and parametricity in session-based communication. In Matthias Felleisen and Philippa Gardner, editors, *Programming Languages and Systems*, Lecture Notes in Computer Science, pages 330–349. Springer Berlin Heidelberg, 2013. ISBN: 978-3-642-37036-6.

[7] Luís Caires and Frank Pfenning. Session types as intuitionistic linear propositions. In Paul Gastin and François Laroussinie, editors, *CONCUR 2010 - Concurrency Theory*, Lecture Notes in Computer Science, pages 222–236. Springer Berlin Heidelberg, 2010. ISBN: 978-3-642-15375-4.

[8] Luís Caires and Hugo Torres Vieira. Conversation types. *Theoretical Computer Science*. European Symposium on Programming 2009, 411(51):4399–4440, December 4, 2010. ISSN: 0304-3975. DOI: 10.1016/j.tcs.2010.09.010.

[9] Sara Capecchi, Elena Giachino, and Nobuko Yoshida. Global escape in multiparty sessions. *Mathematical Structures in Computer Science*, 2016. DOI: 10.1017/S0960129514000164.

[10] Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured interactional exceptions in session types. In Franck van Breugel and Marsha Chechik, editors, *CONCUR 2008 - Concurrency Theory*, Lecture Notes in Computer Science, pages 402–417. Springer Berlin Heidelberg, 2008. ISBN: 978-3-540-85361-9.

[11] Marco Carbone, Nobuko Yoshida, and Kohei Honda. Asynchronous session types: exceptions and multiparty interactions. In Marco Bernardo, Luca Padovani, and Gianluigi Zavattaro, editors, *Formal Methods for Web Services: 9th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2009, Bertinoro, Italy, June 1-6, 2009, Advanced Lectures*, Lecture Notes in Computer Science, pages 187–212. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. ISBN: 978-3-642-01918-0. DOI: 10.1007/978-3-642-01918-0_5.

[12] Alonzo Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58(2):345–363, 1936. ISSN: 0002-9327. DOI: 10.2307/2371045.

[13] Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76(2):95–120, February 1, 1988. ISSN: 0890-5401. DOI: 10.1016/0890-5401(88)90005-3.

[14] Haskell B. Curry. Functionality in combinatory logic. *Proceedings of the National Academy of Sciences of the United States of America*, 20(11):584–590, November 1934. ISSN: 0027-8424.

[15] Haskell B. Curry and Robert Feys. Combinatory logic. North-Holland, 1958.

[16] Nicolaas G. de Bruijn. The mathematical language AUTOMATH, its usage, and some of its extensions. In M. Laudet, D. Lacombe, L. Nolin, and M. Schützenberger, editors, *Symposium on Automatic Demonstration*, Lecture Notes in Mathematics, pages 29–61. Springer Berlin Heidelberg, 1970. ISBN: 978-3-540-36262-3.

[17] Henry DeYoung, Luís Caires, Frank Pfenning, and Bernardo Toninho. Cut reduction in linear logic as asynchronous session-typed communication. In Patrick Cégielski and Arnaud Durand, editors, *Computer Science Logic (CSL'12) - 26th International Workshop/21st Annual Conference of the EACSL*, volume 16 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 228–242, Dagstuhl, Germany. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2012. ISBN: 978-3-939897-42-2. DOI: 10.4230/LIPIcs.CSL.2012.228.

[18] Mariangiola Dezani-Ciancaglini, Sophia Drossopoulou, Dimitris Mostrous, and Nobuko Yoshida. Objects and session types. *Information and Computation*. From Type Theory to Morphological Complexity: Special Issue dedicated to the 60th Birthday Anniversary of Giuseppe Longo, 207(5):595–641, May 1, 2009. ISSN: 0890-5401. DOI: 10.1016/j.ic.2008.03.028.

[19] Simon Fowler, Sam Lindley, J. Garrett Morris, and Sára Decova. Exceptional asynchronous session types: session types without tiers. *Proceedings of the ACM on Programming Languages*, 3:1–29, POPL, January 2, 2019. ISSN: 24751421. DOI: 10.1145/3290341.

[20] Simon J. Gay. Bounded polymorphism in session types. *Mathematical Structures in Computer Science*, 18(5):895–930, October 2008. ISSN: 1469-8072, 0960-1295. DOI: 10.1017/S0960129508006944.

[21] Simon J. Gay and Vasco T. Vasconcelos. Linear type theory for asynchronous session types. *Journal of Functional Programming*, 20(1):19–50, January 2010. ISSN: 1469-7653, 0956-7968. DOI: 10.1017/S0956796809990268.

[22] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–101, 1987. ISSN: 03043975. DOI: 10.1016/0304-3975(87)90045-4.

[23] Jean-Yves Girard and Yves Lafont. Linear logic and lazy computation. In Hartmut Ehrig, Robert Kowalski, Giorgio Levi, and Ugo Montanari, editors, *TAPSOFT '87*, Lecture Notes in Computer Science, pages 52–66. Springer Berlin Heidelberg, 1987. ISBN: 978-3-540-47717-4.

[24] Georges Gonthier. Formal proof—the four-color theorem. *American Mathematical Society*, 55(11):12, 2008.

[25] Daniele Gorla. Towards a unified approach to encodability and separation results for process calculi. *Information and Computation*, 208(9):1031–1053, September 1, 2010. ISSN: 0890-5401. DOI: 10.1016/j.ic.2010.05.002.

[26] Bjørn Haagensen, Sergio Maffeis, and Iain Phillips. Matching systems for concurrent calculi. *Electronic Notes in Theoretical Computer Science*. Proceedings of the 14th International Workshop on Expressiveness in Concurrency (EXPRESS 2007), 194(2):85–99, January 16, 2008. ISSN: 1571-0661. DOI: 10.1016/j.entcs.2007.11.004.

[27] Thomas C. Hales. Historical overview of the kepler conjecture. In Jeffrey C. Lagarias, editor, *The Kepler Conjecture: The Hales-Ferguson Proof*, pages 65–82. Springer New York, New York, NY, 2011. ISBN: 978-1-4614-1129-1. DOI: 10.1007/978-1-4614-1129-1_3.

[28] Thomas C. Hales, Mark Adams, Gertrud Bauer, Tat Dat Dang, John Harrison, Le Truong Hoang, Cezary Kaliszyk, Victor Magron, Sean Mclaughlin, Tat Thang Nguyen, Quang Truong Nguyen, Tobias Nipkow, Steven Obua, Joseph Pleso, Jason Rute, Alexey Solovyev, Thi Hoai An Ta, Nam Trung Tran, Thi Diep Trieu, Josef Urban, Ky Vu, and Roland Zumkeller. A formal proof of the kepler conjecture. *Forum of Mathematics, Pi*, 5, 2017. ISSN: 2050-5086. DOI: 10.1017/fmp.2017.1.

[29] Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In Chris Hankin, editor, *Programming Languages and Systems*, Lecture Notes in Computer Science, pages 122–138. Springer Berlin Heidelberg, 1998. ISBN: 978-3-540-69722-0.

[30] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 273–284, New York, NY, USA. ACM, 2008. ISBN: 978-1-59593-689-9. DOI: 10.1145/1328438.1328472. event-place: San Francisco, California, USA.

[31] William A. Howard. The formulae-as-types notion of construction. In *To H. B. Curry: Essays on combinatory logic, lambda calculus and formalism*, pages 479–491. Academic Press, London, 1980. The original version was circulated privately in 1969.

[32] Raymond Hu, Nobuko Yoshida, and Kohei Honda. Session-based distributed programming in java. In Jan Vitek, editor, *ECOOP 2008 – Object-Oriented Programming*, Lecture Notes in Computer Science, pages 516–541. Springer Berlin Heidelberg, 2008. ISBN: 978-3-540-70592-5.

[33] Hans Hüttel, Ivan Lanese, Vasco T. Vasconcelos, Luís Caires, Marco Carbone, Pierre-Malo Deniélou, Dimitris Mostrous, Luca Padovani, António Ravara, Emilio Tuosto, Hugo Torres Vieira, and Gianluigi Zavattaro. Foundations of session types and behavioural contracts. *ACM Comput. Surv.*, 49(1):3:1–3:36, April 2016. ISSN: 0360-0300. DOI: 10.1145/2873052.

[34] Per Martin-Löf. Intuitionistic type theory. *Bibliopolis Naples, Italy*, 9, 1984.

[35] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I. *Information and Computation*, 100(1):1–40, September 1, 1992. ISSN: 0890-5401. DOI: 10.1016/0890-5401(92)90008-4.

[36] Dimitris Mostrous and Vasco T. Vasconcelos. Affine sessions. In Eva Kühn and Rosario Pugliese, editors, *Coordination Models and Languages*, Lecture Notes in Computer Science, pages 115–130. Springer Berlin Heidelberg, 2014. ISBN: 978-3-662-43376-8.

[37] Catuscia Palamidessi. Comparing the expressive power of the synchronous and the asynchronous pi-calculus. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, pages 256–265, New York, NY, USA. ACM, 1997. ISBN: 978-0-89791-853-4. DOI: 10.1145/263699.263731. event-place: Paris, France.

[38] Davide Sangiorgi and David Walker. *The Pi-Calculus: A Theory of Mobile Processes*. Cambridge University Press, October 16, 2003. 600 pages. ISBN: 978-0-521-54327-9. Google-Books-ID: QkBL_7VtiPgC.

[39] Bernardo Toninho, Luis Caires, and Frank Pfenning. Corecursion and non-divergence in session-typed processes. In Matteo Maffei and Emilio Tuosto, editors, *Trustworthy Global Computing*, Lecture Notes in Computer Science, pages 159–175. Springer Berlin Heidelberg, 2014. ISBN: 978-3-662-45917-1.

[40] Hugo T. Vieira, Luís Caires, and João C. Seco. The conversation calculus: a model of service-oriented computation. In Sophia Drossopoulou, editor, *Programming Languages and Systems*, Lecture Notes in Computer Science, pages 269–283. Springer Berlin Heidelberg, 2008. ISBN: 978-3-540-78739-6.

[41] Philip Wadler. Propositions as types. *Communications of the ACM*, 58(12):75–84, December 2015. ISSN: 0001-0782. DOI: 10.1145/2699407.

Appendices

Appendix A

The CYH type system

For reference, we provide a copy of the type system of CYH, as presented by Carbone et al. in [11, Table 4, p. 197]. We omit typings of terms we do not support in our conversion (cf. Section 3.2), such as recursion and conditionals. In order to keep this thesis self-contained, we stick to the notational conventions from this thesis, rather than those from [11].

$$\frac{}{\Gamma, a : \langle \alpha \rangle \vdash a : \langle \alpha \rangle} \text{ (NAME)} \qquad \frac{\Gamma, a : \alpha\{\![\beta]\!\} \vdash P \triangleright \Delta}{\Gamma \vdash (\nu a)P \triangleright \Delta} \text{ (TRES)}$$

$$\frac{\begin{array}{ll}\Gamma \vdash P \triangleright (x_i : \overline{\alpha_i}\{\![\overline{\beta_i}]\!\})_i & \\ \Gamma' \vdash Q \triangleright (x_i : \overline{\beta_i})_i & s = x_j \\ \Gamma \vdash c : \langle \alpha_j\{\![\beta_j]\!\}\rangle & \Gamma' \subseteq \Gamma, \mathsf{fv}(\Gamma') = \emptyset\end{array}}{\Gamma \vdash \overline{c}(s)[\tilde{x}, P, Q] \triangleright (x_i : \overline{\alpha_i}\{\![\overline{\beta_i}]\!\})_{i \neq j}} \text{ (TREQ)}$$

$$\frac{\begin{array}{ll}\Gamma \vdash P \triangleright s : \alpha\{\![\beta]\!\} & \\ \Gamma \vdash Q \triangleright s : \beta & \mathsf{fv}(\Gamma) = \emptyset\end{array}}{\Gamma, a : \langle \alpha\{\![\beta]\!\} \vdash *a(s)[P, Q] \triangleright \emptyset} \text{ (TSERV)}$$

$$\frac{\mathsf{fv}(\Gamma) = \emptyset}{\Gamma \vdash \mathbf{throw} \triangleright (x_i : \alpha_i)_i} \text{ (TTHR)} \qquad \frac{\Gamma \vdash P_i \triangleright \Delta_i (i = 1, 2) \quad \Delta_1 \asymp \Delta_2}{\Gamma \vdash P_1 \parallel P_2 \triangleright \Delta_1 \cup \Delta_2} \text{ (TPAR)}$$

$$\frac{\Gamma \vdash e : \theta \quad \Gamma \vdash P \triangleright \Delta \cdot x : \alpha}{\Gamma \vdash x!\langle e \rangle.P \triangleright \Delta \cdot x : \uparrow(\theta).\alpha} \text{ (TOUT)} \qquad \frac{\Gamma, y : \theta \vdash P \triangleright \Delta \cdot x : \alpha}{\Gamma \vdash x?(y).P \triangleright \Delta \cdot x : \downarrow(\theta).\alpha} \text{ (TIN)}$$

$$\frac{\Gamma \vdash P_i \triangleright \Delta \cdot x : \alpha_i (\forall i \in I)}{\Gamma \vdash x \rhd \{l_i : P_i\}_{i \in I} \triangleright \Delta \cdot x : \&\{l_i : \alpha_i\}_{i \in I}} \text{ (TBRA)}$$

$$\frac{\Gamma \vdash P \triangleright \Delta \cdot x : \alpha_j \quad j \in I}{\Gamma \vdash x \lhd l_j.P \triangleright \Delta \cdot x : \oplus\{l_i : \alpha_i\}_{i \in I}} \text{ (TSEL)}$$

$$\frac{\mathsf{fv}(\Gamma) = \emptyset \quad \alpha_i \in \{\mathtt{end}, \mathtt{end}\{\![\beta_i]\!\}\}}{\Gamma \vdash \mathbf{0} \triangleright (x_i : \alpha_i)_i} \text{ (TINACT)}$$

Here, Γ is the unrestricted and Δ the linear context. $\mathsf{fv}(\Gamma) = \emptyset$ makes sure Γ contains no free variables, i.e. it prevents certain rules from being used on the service definition level. $\Delta_1 \asymp \Delta_2$ (i.e. Δ_1 and Δ_2 are *compatible*) means that Δ_1 and Δ_2 have no free channel names in common.

Appendix B

Conversion example

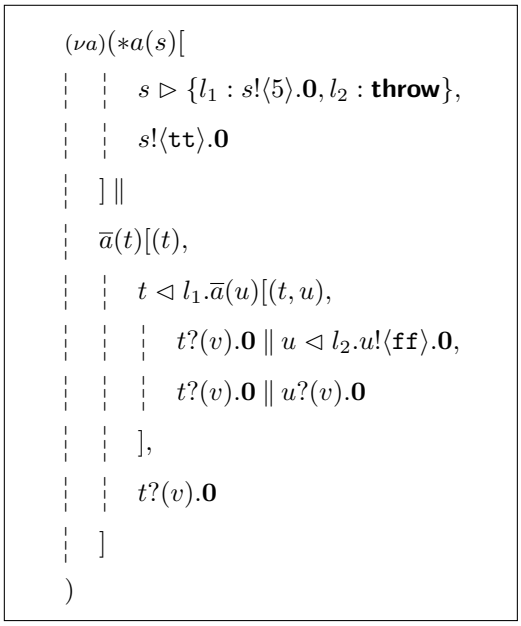


Figure B.1: A CYH$^{\mathsf{CH}}$ process

In order to demonstrate what a converted process looks like, we give an example that showcases the intricacies of the conversion: multiple requests (refinement), parallel processes of different length, and the throwing of exceptions. We will convert the CYH$^{\mathsf{CH}}$ process P, given in Figure B.1, in which we connect to the same service twice.

The converted process $\mathsf{C}(P)$ is given by:

$$(\nu a)(Q \parallel R)$$

See (B.1) for Q and (B.4) for R.

The conversion of the service Q is given by:

$$\begin{aligned} Q = {} & !a(s); (\nu s')(\nu s_1)(\overline{s}\langle s', s_1\rangle \parallel \\ & \quad s'.\mathtt{some}_{\emptyset}(s'_1); Q_{exc} \parallel Q_{serv} \\ &) \end{aligned} \tag{B.1}$$

See (B.2) for Q_{exc} and (B.3) for Q_{serv}.

The exception handler Q_{exc} is given by:

$$Q_{exc} = (\nu v)(\nu s'_2)(\overline{s'_1}\langle v, s'_2\rangle \parallel E_{\mathtt{tt}\to v} \parallel s'_2.\mathtt{close}) \tag{B.2}$$

The service default handler Q_{serv} is given by:

$$
\begin{aligned}
Q_{serv} = {} & s.\mathtt{some}_{\emptyset}(s_1); (\nu s_2)(s_1.\overline{\mathtt{inl}}\langle s_2\rangle \parallel \\
& \quad s_2.\mathtt{some}_{\emptyset}(s_3); s_3.\mathtt{case}(s_4)(\\
& \quad\quad l_1 : s_4.\mathtt{some}_{\emptyset}(s_5); (\nu s_6)(s_5.\overline{\mathtt{inl}}\langle s_6\rangle \parallel \\
& \quad\quad\quad s_6.\mathtt{some}_{\emptyset}(s_7); (\nu v)(\nu s_8)(\overline{s_7}\langle v, s_8\rangle \parallel E_{5\to v} \parallel \\
& \quad\quad\quad\quad s_8.\mathtt{some}_{\emptyset}(s_9); (\nu s_{10})(s_9.\overline{\mathtt{inl}}\langle s_{10}\rangle \parallel \\
& \quad\quad\quad\quad\quad s_{10}.\mathtt{some}_{\emptyset}(s_{11}); s_{11}.\mathtt{close} \\
& \quad\quad\quad\quad) \\
& \quad\quad\quad) \\
& \quad\quad), \\
& \quad\quad l_2 : s_4.\mathtt{some}_{\emptyset}(s_5); (\nu s_6)(s_5.\overline{\mathtt{inr}}\langle s_6\rangle \parallel s_6.\mathtt{close}) \\
& \quad) \\
&)
\end{aligned}
\tag{B.3}
$$

The conversion of the request R is given by:

$$
\begin{aligned}
R = {} & (\nu t)(\overline{a}\langle t\rangle \parallel \\
& \quad t(t', t_1); (\nu y)(y.\mathtt{case}(y_1)(\\
& \quad\quad y_1.\mathtt{some}_{(t')}(y_2); (y_2.\mathtt{close} \parallel \\
& \quad\quad\quad (\nu t_1')(t'.\overline{\mathtt{some}}\langle t_1'\rangle \parallel R_{exc}) \\
& \quad\quad), \\
& \quad\quad (\nu t'')(\nu y_2)(\overline{y_1}\langle t'', y_2\rangle \parallel [t' \leftrightarrow t''] \parallel y_2.\mathtt{close}) \\
& \quad) \parallel R_{req} \\
&)
\end{aligned}
\tag{B.4}
$$

See (B.5) for R_{exc} and (B.6) for R_{req}.

The exception handler R_{exc} is given by:

$$
R_{exc} = t_1'(v, t_2'); (C_v \parallel t_2'.\overline{\mathtt{close}}) \tag{B.5}
$$

The request default handler R_{req} is given by:

$$
\begin{aligned}
R_{req} = {} & (\nu t_2)(t_1.\overline{\texttt{some}}\langle t_2\rangle \parallel \\
& \quad t_2.\texttt{case}(t_3)(\\
& \quad\quad (\nu t_4)(t_3.\overline{\texttt{some}}\langle t_4\rangle \parallel \\
& \quad\quad\quad (\nu t_5)(t_4.\overline{l_1}\langle t_5\rangle \parallel S) \\
& \quad\quad), t_3.\overline{\texttt{close}} \parallel (\nu y_1)(y.\overline{\texttt{inl}}\langle y_1\rangle \parallel \\
& \quad\quad\quad (\nu y_2)(y_1.\overline{\texttt{some}}\langle y_2\rangle \parallel y_2.\overline{\texttt{close}}) \\
& \quad\quad) \\
& \quad) \\
&)
\end{aligned}
\tag{B.6}
$$

See (B.7) for S.

The refinement S is given by:

$$
\begin{aligned}
S = {} & (\nu u)(\overline{a}\langle u\rangle \parallel \\
& \quad (\nu y_1)(y.\overline{\texttt{inr}}\langle y_1\rangle \parallel \\
& \quad\quad y_1(t', y_2); (y_2.\overline{\texttt{close}} \parallel \\
& \quad\quad\quad u(u', u_1); (\nu y')(y'.\texttt{case}(y'_1)(\\
& \quad\quad\quad\quad y'_1.\texttt{some}_{(t',u')}(y'_2); (y'_2.\texttt{close} \parallel \\
& \quad\quad\quad\quad\quad (\nu t'_1)(t'.\overline{\texttt{some}}\langle t'_1\rangle \parallel \\
& \quad\quad\quad\quad\quad\quad (\nu u'_1)(u'.\overline{\texttt{some}}\langle u'_1\rangle \parallel S_{exc}) \\
& \quad\quad\quad\quad\quad), \\
& \quad\quad\quad\quad\quad (\nu t'')(\nu y_2)(\overline{y_1}\langle t'', y_2\rangle \parallel [t' \leftrightarrow t''] \parallel \\
& \quad\quad\quad\quad\quad\quad (\nu u'')(\nu y_3)(\overline{y_2}\langle u'', y_3\rangle \parallel [u' \leftrightarrow u''] \parallel y_3.\texttt{close}) \\
& \quad\quad\quad\quad) \parallel S_{req} \\
& \quad\quad\quad) \\
& \quad\quad) \\
& \quad) \\
&)
\end{aligned}
\tag{B.7}
$$

See (B.8) for S_{exc} and (B.9) for S_{req}.

The exception handler S_{exc} is given by:

$$S_{exc} = t_1'(v, t_2'); (C_v \parallel t_2'.\texttt{close}) \parallel u_1'(v, u_2'); (C_v \parallel u_2'.\texttt{close}) \tag{B.8}$$

The request default handler S_{req} is given by:

$$S_{req} = (\nu z)(S_1 \parallel S_2) \tag{B.9}$$

See (B.10) for S_1 and (B.13) for S_2.

The left process S_1 is given by:

$$\begin{aligned}
S_1 = {} & (\nu t_6)(t_5.\overline{\texttt{some}}\langle t_6\rangle \parallel \\
& \quad t_6.\texttt{case}(t_7)(\\
& \quad\quad (\nu z_1)(z.\overline{\texttt{some}}\langle z_1\rangle \parallel \\
& \quad\quad\quad z_1.\texttt{case}(z_2)(\\
& \quad\quad\quad\quad (\nu t_8)(t_7.\overline{\texttt{some}}\langle t_8\rangle \parallel \\
& \quad\quad\quad\quad\quad (\nu z_3)(z_2.\overline{\texttt{some}}\langle z_3\rangle \parallel \\
& \quad\quad\quad\quad\quad\quad t_8(v, t_9); (C_v \parallel S_1') \\
& \quad\quad\quad\quad\quad) \\
& \quad\quad\quad\quad), z_2.\overline{\texttt{close}} \parallel t_7.\overline{\texttt{none}} \parallel S_{invoke} \\
& \quad\quad\quad) \\
& \quad\quad), t_7.\overline{\texttt{close}} \parallel z.\overline{\texttt{none}} \parallel S_{invoke} \\
& \quad) \\
&)
\end{aligned} \tag{B.10}$$

See (B.11) for S_1' and (B.12) for S_{invoke}.

The continuation of the left process S_1' is given by:

$$
\begin{aligned}
S_1' = {} & (\nu z_4)(z_3.\overline{\texttt{some}}\langle z_4\rangle \parallel \\
& \quad z_4.\texttt{case}(z_5)(\\
& \quad\quad (\nu z_6)(z_5.\overline{\texttt{some}}\langle z_6\rangle \parallel \\
& \quad\quad\quad (\nu t_{10})(t_9.\overline{\texttt{some}}\langle t_{10}\rangle \parallel \\
& \quad\quad\quad\quad t_{10}.\texttt{case}(t_{11})(\\
& \quad\quad\quad\quad\quad (\nu z_7)(z_6.\overline{\texttt{some}}\langle z_7\rangle \parallel \\
& \quad\quad\quad\quad\quad\quad z_7.\texttt{case}(z_8)(\\
& \quad\quad\quad\quad\quad\quad\quad (\nu t_{12})(t_{11}.\overline{\texttt{some}}\langle t_{12}\rangle \parallel t_{12}.\overline{\texttt{close}}) \parallel \\
& \quad\quad\quad\quad\quad\quad\quad (\nu z_9)(z_8.\overline{\texttt{some}}\langle z_9\rangle \parallel z_9.\overline{\texttt{close}}) \parallel \\
& \quad\quad\quad\quad\quad\quad\quad (\nu y_1')(y'.\overline{\texttt{inl}}\langle y_1'\rangle \parallel y_1'.\overline{\texttt{none}}), \\
& \quad\quad\quad\quad\quad\quad\quad z_8.\overline{\texttt{close}} \parallel t_{11}.\overline{\texttt{none}} \parallel S_{invoke} \\
& \quad\quad\quad\quad\quad\quad) \\
& \quad\quad\quad\quad\quad), t_{11}.\overline{\texttt{close}} \parallel z_6.\overline{\texttt{none}} \parallel S_{invoke} \\
& \quad\quad\quad\quad) \\
& \quad\quad\quad) \\
& \quad\quad), z_5.\overline{\texttt{close}} \parallel t_9.\overline{\texttt{none}} \parallel S_{invoke} \\
& \quad) \\
&)
\end{aligned}
\tag{B.11}
$$

See (B.12) for S_{invoke}.

The invocation of the exception handler S_{invoke} is given by:

$$
S_{invoke} = (\nu y_1')(y'.\overline{\texttt{inl}}\langle y_1'\rangle \parallel (\nu y_2')(y_1'.\overline{\texttt{some}}\langle y_2'\rangle \parallel y_2'.\overline{\texttt{close}})) \tag{B.12}
$$

The right process S_2 is given by:

$$
\begin{array}{l}
S_2 = z.\mathtt{some}_{(u_1)}(z_1); (\nu u_2)(u_1.\overline{\mathtt{some}}\langle u_2\rangle \parallel \\
\quad u_2.\mathtt{case}(u_3)(\\
\quad\quad (\nu z_2)(z_1.\overline{\mathtt{inl}}\langle z_2\rangle \parallel \\
\quad\quad\quad z_2.\mathtt{some}_{(u_3)}(z_3); (\nu u_4)(u_3.\overline{\mathtt{some}}\langle u_4\rangle \parallel \\
\quad\quad\quad\quad (\nu u_5)(u_4.\overline{l_2}\langle u_5\rangle \parallel S_2') \\
\quad\quad\quad) \\
\quad\quad), u_3.\overline{\mathtt{close}} \parallel (\nu z_2)(z_1.\overline{\mathtt{inr}}\langle z_2\rangle \parallel z_2.\mathtt{close}) \\
\quad) \\
)
\end{array}
\tag{B.13}
$$

See (B.14) for S_2'.

The continuation of the right process S_2' is given by:

$$
\begin{aligned}
S_2' = {} & z_3.\mathtt{some}_{(u_5)}(z_4); (\nu u_6)(u_5.\overline{\mathtt{some}}\langle u_6\rangle \parallel \\
& u_6.\mathtt{case}(u_7)(\\
& \quad (\nu z_5)(z_4.\overline{\mathtt{inl}}\langle z_5\rangle \parallel \\
& \quad\quad z_5.\mathtt{some}_{(u_7)}(z_6); (\nu u_8)(u_7.\overline{\mathtt{some}}\langle u_8\rangle \parallel \\
& \quad\quad\quad (\nu v)(\nu u_9)(\overline{u_8}\langle v, u_9\rangle \parallel E_{\mathtt{ff}\to v} \parallel \\
& \quad\quad\quad\quad z_6.\mathtt{some}_{(u_9)}(z_7); (\nu u_{10})(u_9.\overline{\mathtt{some}}\langle u_{10}\rangle \parallel \\
& \quad\quad\quad\quad\quad u_{10}.\mathtt{case}(u_{11})(\\
& \quad\quad\quad\quad\quad\quad (\nu z_8)(z_7.\overline{\mathtt{inl}}\langle z_8\rangle \parallel \\
& \quad\quad\quad\quad\quad\quad\quad z_8.\mathtt{some}_{(u_{11})}(z_9); (\\
& \quad\quad\quad\quad\quad\quad\quad\quad (\nu u_{12})(u_{11}.\overline{\mathtt{some}}\langle u_{12}\rangle \parallel u_{12}.\overline{\mathtt{close}}) \parallel \\
& \quad\quad\quad\quad\quad\quad\quad\quad z_9.\mathtt{close} \\
& \quad\quad\quad\quad\quad\quad\quad) \\
& \quad\quad\quad\quad\quad\quad), u_{11}.\overline{\mathtt{close}} \parallel (\nu z_8)(z_7.\overline{\mathtt{inr}}\langle z_8\rangle \parallel z_7.\mathtt{close}) \\
& \quad\quad\quad\quad\quad) \\
& \quad\quad\quad\quad) \\
& \quad\quad\quad) \\
& \quad\quad), u_7.\overline{\mathtt{close}} \parallel (\nu z_5)(z_4.\overline{\mathtt{inr}}\langle z_5\rangle \parallel z_5.\mathtt{close}) \\
& \quad) \\
&)
\end{aligned}
\tag{B.14}
$$

Appendix C

Synchronization diagrams

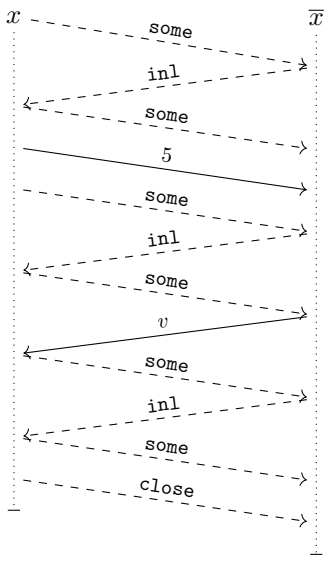
This appendix contains several example $\mathsf{CYH}^{\mathsf{CH}}$ processes, complemented with diagrams showing the synchronizations in the converted DCPTP process. This should give some insight in the workings of the conversion given in Section 3.3.

We only give the requests of our example processes. In the diagrams, we indicate the server side of a channel with an overbar (so $\overline{x}$ where x is used in the request). Dashed lines indicate synchronization messages, and solid lines indicate the actual communication.

Example 1

This example shows a simple protocol that successfully performs two communications.

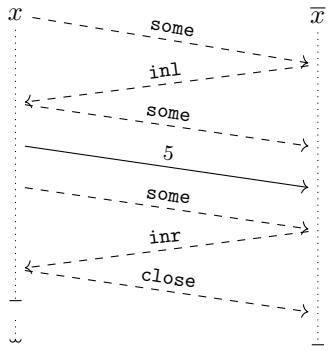
$$\overline{a}(x)[(x), x!\langle 5\rangle.x?(v).\mathbf{0}, Q^{(x)}]$$



Example 2

This example shows the same protocol as in the previous example, except that an exception is thrown by the server after the first communication succeeds.

$$\overline{a}(x)[(x), x!\langle 5\rangle.x?(v).\mathbf{0}, Q^{(x)}]$$

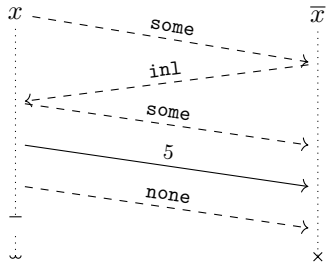


invoke exception handler

Example 3

This example shows the same protocol as in the previous examples, except that it throws an exception after the first communication succeeds.

$$\overline{a}(x)[(x), x!\langle 5\rangle.\textsf{throw}, Q^{(x)}]$$

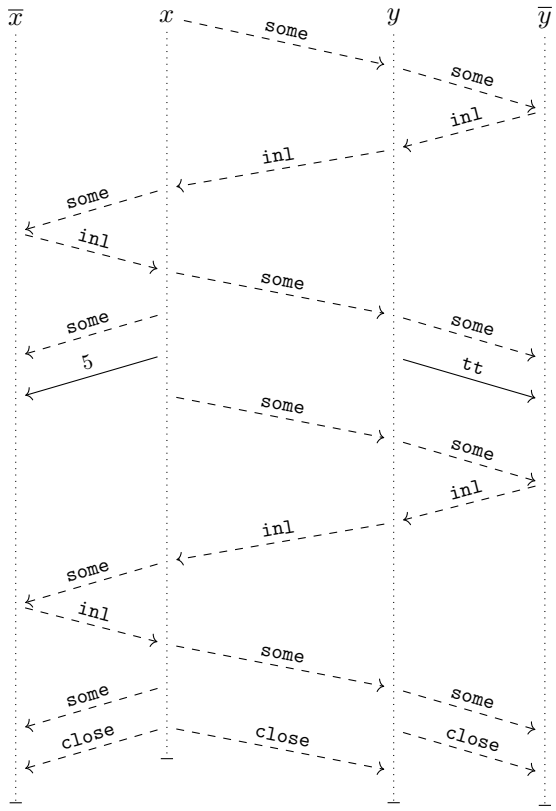


invoke exception handler

Example 4

This example shows a protocol that connects to two services and then performs a single communication on each connected channel in parallel. All communications succeed.

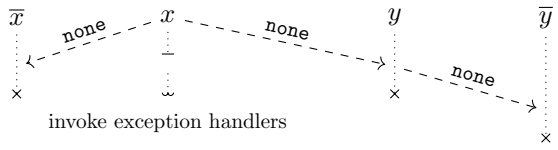
$$\overline{a}(x)[(x), \overline{b}(y)[(x, y), x!\langle 5\rangle.\mathbf{0} \parallel y!\langle \mathtt{tt}\rangle.\mathbf{0}, Q^{(x,y)}], R^{(x)}]$$



Example 5

This example is similar to the previous one, except that an exception is thrown on the left side of the parallel composition.

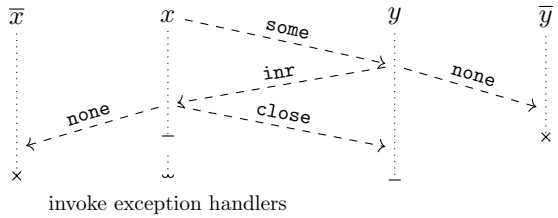
$$\overline{a}(x)[(x), \overline{b}(y)[(x,y), \textbf{throw} \parallel y!\langle \texttt{tt} \rangle.\mathbf{0}, Q^{(x,y)}], R^{(x)}]$$



Example 6

Again, this example is similar to the previous two, except now the exception is thrown on the right side of the parallel composition.

$$\overline{a}(x)[(x), \overline{b}(y)[(x,y), x!\langle 5 \rangle.\mathbf{0} \parallel \textbf{throw}, Q^{(x,y)}], R^{(x)}]$$



Appendix D

A multiparty calculus

In this appendix, we describe a multiparty calculus, based on the original calculus by Honda et al. [30] and the slightly more expressive version by Caires and Pérez [5], extended with a global operator for non-determinism. We include a projection into our binary system DCPTP, given in Chapter 2.

D.1 Global and local types

We define global types (G), base types (U) and local types (T). Here, $p, q, r, \ldots$ are participants, and $l_1, l_2, \ldots$ are labels. At the end of this section, on page 82 we explain the design choice of the new global type this thesis introduces: $p \,\&\, G$.

Definition D.1 (Global and local types)

Global types (G) and local types (T) are given by

$$G ::= \mathtt{end} \mid p \twoheadrightarrow q : \{l_i\langle U_i\rangle : G_i\}_{i\in I} \mid (G_1 \parallel G_2) \mid p \,\&\, G$$
$$U ::= \mathsf{bool} \mid \mathsf{nat} \mid \mathsf{str} \mid \cdots \mid T$$
$$T ::= \mathtt{end} \mid p?\{l_i\langle U_i\rangle : T_i\}_{i\in I} \mid p!\{l_i\langle U_i\rangle : T_i\}_{i\in I} \mid \&T \mid \oplus T$$

A global type G describes a sequence of interactions between two or more participants. The first three types are as usual, taken from [30, 5]. The last type is newly added in this thesis.

- `end` indicates the end of communication for all participants.
- $(G_1 \parallel G_2)$ puts two independent global types in parallel. Here, independence means that G_1 and G_2 have no participants in common. The participants of a global type are defined in detail below.
- $p \twoheadrightarrow q : \{l_i\langle U_i\rangle : G_i\}_{i\in I}$ signifies a communication between p and q. Participant p will choose between a set of labels l_i for $i \in I$ (cf. branching in Chapter 2). A choice for l_i means p will send a message of type U_i to q, after which the global session continues as G_i.

- $p \,\&\, G$ gives participant p the choice to continue the global session as G, or to abort all communications. Here, the word "choice" does not induce determinism: p's choice can depend on external factors such as resource availability or the possibility of an operation to fail.

Local types T represent local views of the global type for a specific participant. They can be obtained from a global type by means of projection. Before explaining local types, it is useful to define the participants of a global type.

Definition D.2 (Participants)

Given a global type G, the set of participants $\mathsf{part}(G)$ is given recursively by

- $\mathsf{part}(\mathtt{end}) = \emptyset$
- $\mathsf{part}(G_1 \parallel G_2) = \mathsf{part}(G_1) \cup \mathsf{part}(G_2)$
- $\mathsf{part}(p \twoheadrightarrow q : \{l_i\langle U_i\rangle : G_i\}_{i\in I}) = \{p, q\} \cup \bigcup_{i\in I} \mathsf{part}(G_i)$
- $\mathsf{part}(p \,\&\, G) = \{p\} \cup \mathsf{part}(G)$

For a clear presentation, we assume the following types are projections under some participant p.

- $\mathtt{end}$ ends the session for p. This type is similar to the global one. However, globally, $\mathtt{end}$ indicates the end of communications for all participants, while locally it only ends communications for p.
- $p!\{l_i\langle U_i\rangle : T_i\}_{i\in I}$ means p will choose a label l_i, send a message of appropriate type U_i, and continue as T_i. We will see later that the lack of a receiving participant does not make the type ambiguous.
- $q?\{l_i\langle U_i\rangle : T_i\}_{i\in I}$ says that p will receive a choice from q for some label l_i, which will then send a message of type U_i to p. After receipt, p continues as T_i.
- $\&T$ allows p to choose (possibly non-deterministically) to continue as T or to abort.
- $\oplus T$ signifies that p will receive a message saying whether they may continue as T or whether they must abort.

The projection of a global type under a participant uses an auxiliary operator which merges two base types. This allows us to define the projection of a branch under a participant that is not involved in the transaction, e.g. the projection of type $p \twoheadrightarrow q : \{\dots\}$ under some participant r.

Definition D.3 (Merge)

The merge operation $\sqcup$ on types U is given by

1. $\mathsf{bool} \sqcup \mathsf{bool} = \mathsf{bool}$ (and similarly for other primitive types)
2. $\mathtt{end} \sqcup \mathtt{end} = \mathtt{end}$
3. $p!\{l_i\langle U_i\rangle : T_i\}_{i\in I} \sqcup p!\{l_i\langle U_i\rangle : T_i\}_{i\in I} = p!\{l_i\langle U_i\rangle : T_i\}_{i\in I}$
4. $p?\{l_k\langle U_k\rangle : T_k\}_{k\in K} \sqcup p?\{l'_j\langle U'_j\rangle : T'_j\}_{j\in J} = p?\{l_k\langle U_k\rangle : T_k\}_{k\in K\setminus J} \cup p?\{l'_j\langle U'_j\rangle : T'_j\}_{j\in J\setminus K} \cup p?\{l_i\langle U_i \sqcup U'_i\rangle : T_i \sqcup T'_i\}_{i\in K\cap J}$
5. $\&T_1 \sqcup \&T_2 = \&(T_1 \sqcup T_2)$
6. $\oplus T_1 \sqcup \oplus T_2 = \oplus(T_1 \sqcup T_2)$

If types do not match any of the above clauses, their merge is undefined.

Two types U_1 and U_2 are *mergeable* if their merge $U_1 \sqcup U_2$ is defined. Rules 1–3 are taken exactly from the original definition of merge in [30]. Rule 4 is taken from [5]. The original of this rule only permits identical options when receiving, while here we allow different options, as long as the options with the same labels are mergeable. Rules 5 and 6 are new and straightforward: inner types need to be mergeable for non-deterministic types to be mergeable.

Definition D.4 (Projection)

Given a global type G and participant r, the projection $G \upharpoonright r$ of G under r is given by

1. $\mathtt{end} \upharpoonright r = \mathtt{end}$
2. $p \twoheadrightarrow q : \{l_i\langle U_i\rangle : G_i\}_{i\in I} \upharpoonright r = \begin{cases} p!\{l_i\langle U_i\rangle : G_i \upharpoonright r\}_{i\in I} & \text{if } r = p \\ p?\{l_i\langle U_i\rangle : G_i \upharpoonright r\}_{i\in I} & \text{if } r = q \\ \bigsqcup_{i\in I} G_i \upharpoonright r & \text{otherwise} \end{cases}$
3. $(G_1 \parallel G_2) \upharpoonright r = \begin{cases} G_1 \upharpoonright r & \text{if } r \in \mathsf{part}(G_1) \text{ and } r \notin \mathsf{part}(G_2) \\ G_2 \upharpoonright r & \text{if } r \notin \mathsf{part}(G_1) \text{ and } r \in \mathsf{part}(G_2) \\ \mathtt{end} & \text{if } r \notin \mathsf{part}(G_1) \cup \mathsf{part}(G_2) \end{cases}$
4. $p \,\&\, G \upharpoonright r = \begin{cases} \&(G \upharpoonright r) & \text{if } r = p \\ \oplus(G \upharpoonright r) & \text{if } r \neq p \text{ and } r \in \mathsf{part}(G) \\ \mathtt{end} & \text{otherwise} \end{cases}$

If a side-condition does not hold, the map is undefined.

Rules 1–3 are taken from [30], but use the merge operation from [5]. Rule 2 shows that a branching type can be projected under a participant either when they are partaking in the transaction, or when they are a participant in all subtypes G_i (due to mergeability). Rule 3 shows that parallel global types can only be projected if they do not share participants. Rule 4 is new in this thesis, albeit straightforward.

Not all global types are useful or flawless. The notion of projection to local types provides a means to characterize the well-formedness of global types. This definition is justified by the characterization results presented in Section D.3.

Definition D.5 (Well-formed global types)

A global type G is well-formed (WF) if the projection $G \upharpoonright r$ is defined for all participants $r \in \mathsf{part}(G)$.

We will see later that it is useful to use the term "Merge well-formedness" for well-formed global types, since it is based on projection using the merge operation. So, we say a global type is merge well-formed (MWF) if and only if it is well-formed (WF).

D.1.1 Discussion: non-deterministic global type

We discuss our choice for the global type with non-determinism $p\&G$. The first question that comes to mind when considering this global type for non-determinism is: why must there be a participant that decides? The reason is that global sessions contain no information about the implementation. Therefore, it is possible to consider a situation where participants are directly connected to each other. In this situation, there is no global party that can independently decide whether to continue or to abort. I.e., the responsibility is with the participants.

Then, why can only one participant decide at a time? It would be possible to give the responsibility to a set of (one or more) participants. We try to add concurrency where possible, so we would like those participants to announce their resource availability in any order. However, we also keep in mind that we want to eventually convert this global type to the binary system, in which we have to use rules such as ($\mathrm{T}\oplus^{x}_{\tilde{w}}$). The process reduction associated with this rule discards the process ($\mathrm{T}\oplus^{x}_{\tilde{w}}$) prefixes, and the channels $\tilde{w}$ can and must be used only in that prefixed process. This makes it impossible to compose those prefixed process using ($\mathrm{T}\|$), while still assuring that the rest of the global session will be discarded if a participant announces unavailability. Hence, there would have to be a pre-determined order for the participants to decide, e.g. $\{p, q\}\,\&\,G \neq \{q, p\}\,\&\,G$.

Let us design a potential non-deterministic global type with ordered consensus: $\{p_1, \ldots, p_k\}\,\&\,G$ with $\mathsf{part}(G) = \{p_1, \ldots, p_k, p_{k+1}, \ldots, p_n\}$. Then, for any $1 \leq i \leq k$,

$$(\{p_1, \ldots, p_k\}\,\&\,G) \upharpoonright p_i = \oplus^{i-1}\,\&\,\oplus^{k-i}(G \upharpoonright p_i)$$

and for any $k < j \leq n$,

$$(\{p_1, \ldots, p_k\}\,\&\,G) \upharpoonright p_j = \oplus(G \upharpoonright p_j).$$

Actually, it is possible to model this type using the existing single-participant type:

$$\{p_1, \ldots, p_k\}\,\&\,G \triangleq p_1\,\&\,\ldots p_k\,\&\,G$$

Only for large k and n is the first explicit consensus type more efficient than the modelling, because then those participants not involved in the consensus (so the p_j where $k < j \leq n$) would only have to receive an availability message once, whereas in the modelled version they would have to receive a message from every participant involved in consensus. However, k and n are not likely to be sufficiently large for this to be an issue. If such a situation does occur, it is always possible to tweak the calculus to use the more efficient definition above.

D.2 Mediums and binary local types

In order to study the multiparty types from the previous section in a binary setting, we define a conversion, based on [5]. The conversion from local types to binary types is straightforward.

Definition D.6 (Binary local types)

The mapping $\langle\!\langle \cdot \rangle\!\rangle$ from local types to binary types is given by

1. $\langle\!\langle \mathsf{end} \rangle\!\rangle = \mathbf{1}$
2. $\langle\!\langle \mathsf{bool} \rangle\!\rangle = \mathbf{1}$ (and similarly for other primitive types)
3. $\langle\!\langle p!\{l_i\langle U_i\rangle : T_i\}_{i\in I} \rangle\!\rangle = \oplus\{l_i : \underline{\langle\!\langle U_i \rangle\!\rangle} \otimes \langle\!\langle T_i \rangle\!\rangle\}_{i\in I}$
4. $\langle\!\langle p?\{l_i\langle U_i\rangle : T_i\}_{i\in I} \rangle\!\rangle = \&\{l_i : \langle\!\langle U_i \rangle\!\rangle \parr \langle\!\langle T_i \rangle\!\rangle\}_{i\in I}$
5. $\langle\!\langle \& T \rangle\!\rangle = \& \langle\!\langle T \rangle\!\rangle$
6. $\langle\!\langle \oplus T \rangle\!\rangle = \oplus \langle\!\langle T \rangle\!\rangle$

This definition showcases an important difference between the intuitionistic and classical systems. Here, rules 3 and 4 exhibit a duality in the type of message that is being exchanged. This makes sense: the sending party has one end of a channel, and sends the other end (of dual type) to the receiving party. In the intuitionistic system (cf. [5]), duality is achieved by placing judgements on the left and the right of the process term, with corresponding (dual) left and right rules. Therefore, the converted intuitionistic binary local types have no duality in the types of the messages.

In order to compose the binary versions of participants, a medium process is extracted from the global type. This process acts as a message router, forwarding messages and choices between participants. The medium process is defined using a mapping $\mathcal{C}$ from participants to channel names.

In order to keep things readable, we do not use common function notation (e.g. $\mathcal{C}(p)$), but we write $\mathcal{C}_p$ to denote the channel name for p. The mapping is assumed to have a unique channel name for each participant: for every $p \neq q$, $\mathcal{C}_p \neq \mathcal{C}_q$. Also, using the mapping with a prime creates a new unique channel name for the given participant: $\mathcal{C}_p \neq \mathcal{C}'_p$. However, these primed channel names are always the same for the same amount of primes and the same participants. The notation $\mathcal{C}_{\{x \mapsto \mathcal{C}'_x\}_{x\in X}}$ transforms $\mathcal{C}$ to use the name from $\mathcal{C}'$ for each participant in X.

Definition D.7 (Mediums)

Given a global type G and mapping $\mathcal{C}$ from participants to channel names, the medium $\mathsf{M}[\![G]\!]_{\mathcal{C}}$ is given by

- $\mathsf{M}[\![\mathtt{end}]\!]_{\mathcal{C}} = \mathbf{0}$
- $\mathsf{M}[\![(G_1 \parallel G_2)]\!]_{\mathcal{C}} = (\mathsf{M}[\![G_1]\!]_{\mathcal{C}} \parallel \mathsf{M}[\![G_2]\!]_{\mathcal{C}})$
- $\mathsf{M}[\![p \twoheadrightarrow q : \{l_i\langle U_i\rangle : G_i\}_{i\in I}]\!]_{\mathcal{C}} =$
  $\mathcal{C}_p.\mathtt{case}(\mathcal{C}'_p)($
  $\quad l_i : \mathcal{C}'_p(u, \mathcal{C}''_p); (\nu\mathcal{C}'_q)($
  $\quad\quad \mathcal{C}_q.\overline{l_i}\langle\mathcal{C}'_q\rangle \parallel (\nu v)(\nu\mathcal{C}''_q)($
  $\quad\quad\quad \overline{\mathcal{C}'_q}\langle v, \mathcal{C}''_q\rangle \parallel [u \leftrightarrow v] \parallel \mathsf{M}[\![G_i]\!]_{\mathcal{C}\{p\mapsto\mathcal{C}''_p, q\mapsto\mathcal{C}''_q\}}$
  $\quad\quad)$
  $\quad)$
  $)_{i\in I}$
- $\mathsf{M}[\![p \,\&\, G]\!]_{\mathcal{C}} = \mathcal{C}_p.\mathtt{some}_{\tilde{\mathcal{C}}_{\mathsf{part}(G)\setminus\{p\}}}(\mathcal{C}'_p);$
  $\quad (\nu\mathcal{C}'_q)_{q\in\mathsf{part}(G)\setminus\{p\}} \left(\prod_{q\in\mathsf{part}(G)\setminus\{p\}} \mathcal{C}_q.\overline{\mathtt{some}}\langle\mathcal{C}'_q\rangle \parallel \mathsf{M}[\![G]\!]_{\mathcal{C}\{q\mapsto\mathcal{C}'_q\}_{q\in\mathsf{part}(G)}}\right),$
  where
  - $(\nu\mathcal{C}_x)_{x\in\{p,q,\ldots,r\}}$ means $(\nu\mathcal{C}_p)(\nu\mathcal{C}_q)\ldots(\nu\mathcal{C}_r)$, and
  - $\prod_{x\in\{p,q,\ldots,r\}} P$ means $P\{p/x\} \parallel P\{q/x\} \parallel \ldots \parallel P\{r/x\}$.

With the medium process in place, we can hint at the characterization results obtained from above conversion. The following definition gives a general notion of when a global type is useful and correct, using the binary calculus.

Definition D.8 (Compositional typing)

The judgement $\mathsf{M}[\![G]\!]_{\mathsf{npart}(G)} \vdash \Delta; \Theta$ is a *compositional typing* if i) it is a valid typing according to Chapter 2, and ii) $\mathsf{npart}(G) \subseteq \mathrm{Dom}(\Delta)$.

In this definition, $\mathsf{npart}(G)$ is the initial map of participants to unique channel names. We formalize this in Definition D.9.

Definition D.9 (Participant names)

Given a global type G, $\mathsf{npart}(G)$ maps participants of G to unique channel names. For any participant r, this map is given by

- $\mathsf{npart}(G_1 \parallel G_2)(r) = \begin{cases} \mathsf{npart}(G_1)(r) & \text{if } r \in \mathsf{part}(G_1) \text{ and } r \notin \mathsf{part}(G_2) \\ \mathsf{npart}(G_2)(r) & \text{if } r \notin \mathsf{part}(G_1) \text{ and } r \in \mathsf{part}(G_2) \end{cases}$
- $\mathsf{npart}(p \twoheadrightarrow q : \{l_i\langle U_i\rangle : G_i\}_{i\in I})(r) = \begin{cases} c_p & \text{if } r = p \\ c_q & \text{if } r = q \\ \mathsf{npart}(\bigsqcup_{i\in I} G_i)(r) & \text{otherwise} \end{cases}$
- $\mathsf{npart}(p \,\&\, G)(r) = \begin{cases} c_p & \text{if } r = p \\ \mathsf{npart}(G)(r) & \text{otherwise} \end{cases}$

D.3 Characterization results

In this section, we describe how well global and binary types are related through the definitions from the previous section. We will show that the binary local views of a global protocol validly type the respective medium process. We also show that a valid typing of a medium process induces local types of the respective global protocol (albeit in a slightly weaker manner). These results are versions of the results from [5] modified for our type systems.

D.3.1 Simple well-formedness

Definition D.4 describes the extraction of local types from global types using the merge operation (in Definition D.3). The usage of this operation adds complexity to the proofs, which makes clear presentation more difficult. Therefore, we first state and prove the results for a simpler version of local projection (hence *simple projection*). This projection requires the simple projections of global types to be the same across branches for any participants not involved in the transaction. Then, we restate the results using the projection with merge, and explain how the simple proofs can be modified for this more complex situation. This procedure follows the proof method for the characterization results of [5].

Definition D.10 (Simple projection)

Given a global type G and participant r, the simple projection $G \wr r$ is exactly as Definition D.4, with the exception of the following case:

$$p \twoheadrightarrow q : \{l_i\langle U_i\rangle : G_i\}_{i\in I} \wr r = \begin{cases} p!\{l_i\langle U_i\rangle : G_i \wr r\}_{i\in I} & \text{if } r = p \\ p?\{l_i\langle U_i\rangle : G_i \wr r\}_{i\in I} & \text{if } r = q \\ G_1 \wr r & \text{if } r \neq p,\ r \neq q, \text{ and} \\ & G_i \wr r = G_j \wr r \text{ for every } i, j \in I \end{cases}$$

This simple interpretation requires a modified notion of well-formedness.

Definition D.11 (Simply well-formed global types)

A global type G is simply well-formed (SWF) if the simple projection $G \wr r$ is defined for all participants $r \in \mathsf{part}(G)$.

The first simple characterization result states that the binary simple projections of a simply well-formed global type validly type the respective medium process.

Theorem D.1

Let G be a global type with $\mathsf{part}(G) = \{p_1, \ldots, p_n\}$ and let $\mathcal{C} = \mathsf{npart}(G)$. If G is SWF, then

$$\mathsf{M}[\![G]\!]_{\mathcal{C}} \vdash \mathcal{C}_{p_1} : \overline{\langle\!\langle G \wr p_1 \rangle\!\rangle}, \ldots, \mathcal{C}_{p_n} : \overline{\langle\!\langle G \wr p_n \rangle\!\rangle}; \Theta$$

is a compositional typing for some persistent context Θ.

We need an auxiliary lemma to prove this theorem. It shows that if simple well-formedness holds for a global type, then it also holds for all its subtypes.

Lemma D.2

i) If $p \twoheadrightarrow q : \{l_i\langle U_i\rangle : G_i\}_{i \in I}$ is SWF, then for all $i \in I$, G_i is SWF too, ii) if $(G_1 \parallel G_2)$ is SWF, then G_1 and G_2 are SWF too, and iii) if $p \,\&\, G$ is SWF, then G is SWF too.

Proof. This follows by construction from Definitions D.11 and D.10. □

Now we are ready to prove Theorem D.1.

Proof. By induction on the structure of G.

- ($G = \mathtt{end}$) We have $\mathsf{M}[\![\mathtt{end}]\!]_{\mathcal{C}} = \mathbf{0}$. By the Empty Axiom,

  $$\mathbf{0} \vdash \cdot; \Theta$$

  is a valid typing. Since the empty linear context coincides with the lack of participants ($\mathsf{part}(G) = \emptyset$), we can conclude that this is a compositional typing.

- ($G = G_1 \parallel G_2$) As in [5], through Lemma D.2 and using the Mix Rule.

- ($G = p \twoheadrightarrow q : \{l_i\langle U_i\rangle : G_i\}_{i \in I}$) W.l.o.g. we assume $p = p_1$ and $q = p_2$. Since we assume G is SWF, all local types $G \wr p_1$, …, $G \wr p_n$ are defined. We have:

  $$G \wr p = p!\{l_i\langle U_i\rangle : G_i \wr p\}_{i \in I} \tag{D.1}$$
  $$G \wr q = p?\{l_i\langle U_i\rangle : G_i \wr q\}_{i \in I} \tag{D.2}$$
  $$G \wr p_j = G_1 \wr p_j \text{ for every } j \in \{3, \ldots, n\}$$

From (D.1) and (D.2) we find the duals of the binary types of p and q:

$$\begin{aligned}\overline{\langle\!\langle G \wr p \rangle\!\rangle} &= \overline{\oplus\{l_1 : \langle\!\langle U_1 \rangle\!\rangle \otimes \langle\!\langle G_1 \wr p \rangle\!\rangle, l_2 : \langle\!\langle U_2 \rangle\!\rangle \otimes \langle\!\langle G_2 \wr p \rangle\!\rangle\}} \\ &= \&\{l_1 : \overline{\langle\!\langle U_1 \rangle\!\rangle} \parr \overline{\langle\!\langle G_1 \wr p \rangle\!\rangle}, l_2 : \overline{\langle\!\langle U_2 \rangle\!\rangle} \parr \overline{\langle\!\langle G_2 \wr p \rangle\!\rangle}\} \qquad \text{(D.3)} \\ \overline{\langle\!\langle G \wr q \rangle\!\rangle} &= \overline{\&\{l_1 : \overline{\langle\!\langle U_1 \rangle\!\rangle} \parr \langle\!\langle G_1 \wr q \rangle\!\rangle, l_2 : \overline{\langle\!\langle U_2 \rangle\!\rangle} \parr \langle\!\langle G_2 \wr q \rangle\!\rangle\}} \\ &= \oplus\{l_1 : \langle\!\langle U_1 \rangle\!\rangle \otimes \overline{\langle\!\langle G_1 \wr q \rangle\!\rangle}, l_2 : \langle\!\langle U_2 \rangle\!\rangle \otimes \overline{\langle\!\langle G_2 \wr q \rangle\!\rangle}\} \qquad \text{(D.4)}\end{aligned}$$

W.l.o.g. we assume $I = \{1, 2\}$. Now, the medium process of G is

$$\mathsf{M}[\![G]\!]_{\mathcal{C}} = \mathcal{C}_p.\mathsf{case}(\mathcal{C}'_p)(l_1 : L_1, l_2 : L_2), \qquad \text{(D.5)}$$

where

$$\begin{aligned}L_1 =& \mathcal{C}'_p(u, \mathcal{C}''_p); (\nu \mathcal{C}'_q)(\mathcal{C}_q.\overline{l_1}\langle \mathcal{C}'_q \rangle \parallel \\ & (\nu v)(\nu \mathcal{C}''_q)(\overline{\mathcal{C}'_q}\langle v, \mathcal{C}''_q \rangle \parallel [u \leftrightarrow v] \parallel \mathsf{M}[\![G_1]\!]_{\mathcal{C}\{p \mapsto \mathcal{C}''_p, q \mapsto \mathcal{C}''_q\}})) \qquad \text{(D.6)}\end{aligned}$$

and

$$\begin{aligned}L_2 =& \mathcal{C}'_p(u, \mathcal{C}''_p); (\nu \mathcal{C}'_q)(\mathcal{C}_q.\overline{l_2}\langle \mathcal{C}'_q \rangle \parallel \\ & (\nu v)(\nu \mathcal{C}''_q)(\overline{\mathcal{C}'_q}\langle v, \mathcal{C}''_q \rangle \parallel [u \leftrightarrow v] \parallel \mathsf{M}[\![G_2]\!]_{\mathcal{C}\{p \mapsto \mathcal{C}''_p, q \mapsto \mathcal{C}''_q\}})). \qquad \text{(D.7)}\end{aligned}$$

Since we assume G is SWF, we know by Lemma D.2 that subtypes G_1 and G_2 are also SWF. This means we can apply the induction hypothesis to infer that

$$\mathsf{M}[\![G_1]\!]_{\mathcal{C}''} \vdash \mathcal{C}''_p : \langle\!\langle G_1 \wr p \rangle\!\rangle, \mathcal{C}''_q : \langle\!\langle G_1 \wr q \rangle\!\rangle, \underbrace{\mathcal{C}''_{p_3} : \langle\!\langle G_1 \wr p_3 \rangle\!\rangle, \ldots, \mathcal{C}''_{p_n} : \langle\!\langle G_1 \wr p_n \rangle\!\rangle}_{\Delta_1}; \Theta$$

and

$$\mathsf{M}[\![G_2]\!]_{\mathcal{C}''} \vdash \mathcal{C}''_p : \langle\!\langle G_2 \wr p \rangle\!\rangle, \mathcal{C}''_q : \langle\!\langle G_2 \wr q \rangle\!\rangle, \underbrace{\mathcal{C}''_{p_3} : \langle\!\langle G_2 \wr p_3 \rangle\!\rangle, \ldots, \mathcal{C}''_{p_n} : \langle\!\langle G_2 \wr p_n \rangle\!\rangle}_{\Delta_2}; \Theta$$

are compositional typings for some Θ, where $\mathcal{C}'' \triangleq \mathcal{C}\{p \mapsto \mathcal{C}''_p, q \mapsto \mathcal{C}''_q\}$.

Figure D.1 shows the typing inference of L_1 (given in (D.6)). Analogously, we may derive the typing of L_2 (given in (D.7)). We find:

$$\begin{aligned}L_1 \vdash & \mathcal{C}'_p : \overline{\langle\!\langle U_1 \rangle\!\rangle} \parr \overline{\langle\!\langle G_1 \wr p \rangle\!\rangle}, \\ & \mathcal{C}_q : \oplus\{l_1 : \langle\!\langle U_1 \rangle\!\rangle \otimes \overline{\langle\!\langle G_1 \wr q \rangle\!\rangle}, l_2 : \langle\!\langle U_2 \rangle\!\rangle \otimes \overline{\langle\!\langle G_2 \wr q \rangle\!\rangle}\}, \Delta_1; \Theta \qquad \text{(D.8)} \\ L_2 \vdash & \mathcal{C}'_p : \overline{\langle\!\langle U_2 \rangle\!\rangle} \parr \overline{\langle\!\langle G_2 \wr p \rangle\!\rangle}, \\ & \mathcal{C}_q : \oplus\{l_1 : \langle\!\langle U_1 \rangle\!\rangle \otimes \overline{\langle\!\langle G_1 \wr q \rangle\!\rangle}, l_2 : \langle\!\langle U_2 \rangle\!\rangle \otimes \overline{\langle\!\langle G_2 \wr q \rangle\!\rangle}\}, \Delta_2; \Theta \qquad \text{(D.9)}\end{aligned}$$

By Definition D.10, for all $j \in \{3, \ldots, n\}$, $G_1 \upharpoonright p_j = G_2 \upharpoonright p_j$, so $\Delta_1 = \Delta_2$. Now we can apply the n-ary Offer Rule with (D.8) and (D.9) as premises to obtain a compositional typing for $\mathsf{M}[\![G]\!]_{\mathcal{C}}$ (cf. (D.5)):

$$\begin{aligned} &\mathcal{C}_p.\mathtt{case}(\mathcal{C}'_p)(l_1 : L_1, l_2 : L_2) \\ \vdash \mathcal{C}_p : &\underbrace{\&\{l_1 : \overline{\langle\!\langle U_1 \rangle\!\rangle} \parr \overline{\langle\!\langle G_1 \upharpoonright p \rangle\!\rangle}, l_2 : \overline{\langle\!\langle U_2 \rangle\!\rangle} \parr \overline{\langle\!\langle G_2 \upharpoonright p \rangle\!\rangle}\}}_{\overline{\langle\!\langle G \upharpoonright p \rangle\!\rangle} \text{ (cf. (D.3))}}, \\ \mathcal{C}_q : &\underbrace{\oplus\{l_1 : \langle\!\langle U_1 \rangle\!\rangle \otimes \overline{\langle\!\langle G_1 \upharpoonright q \rangle\!\rangle}, l_2 : \langle\!\langle U_2 \rangle\!\rangle \otimes \overline{\langle\!\langle G_2 \upharpoonright q \rangle\!\rangle}\}}_{\overline{\langle\!\langle G \upharpoonright q \rangle\!\rangle} \text{ (cf. (D.4))}}, \Delta_1 = \Delta_2; \Theta \end{aligned}$$

- $(G = p \,\&\, G')$ W.l.o.g. we say p for p_1. Since we assume G is SWF, all local types are defined:

$$G \upharpoonright p = \&(G' \upharpoonright p) \tag{D.10}$$
$$G \upharpoonright p_j = \oplus(G' \upharpoonright p_j) \text{ for every } j \in \{2, \ldots, n\} \tag{D.11}$$

From (D.10) and (D.11) we find the dual binary types of G's participants:

$$\begin{aligned} \overline{\langle\!\langle G \upharpoonright p \rangle\!\rangle} &= \overline{\&\langle\!\langle G' \upharpoonright p \rangle\!\rangle} \\ &= \oplus\overline{\langle\!\langle G' \upharpoonright p \rangle\!\rangle} \qquad (D.12) \\ \overline{\langle\!\langle G \upharpoonright p_j \rangle\!\rangle} &= \overline{\oplus\langle\!\langle G' \upharpoonright p_j \rangle\!\rangle} \\ &= \&\overline{\langle\!\langle G' \upharpoonright p_j \rangle\!\rangle} \text{ for every } j \in \{2, \ldots, n\} \qquad (D.13) \end{aligned}$$

The medium process of G is:

$$\mathsf{M}[\![G]\!]_{\mathcal{C}} = \mathcal{C}_p.\mathtt{some}_{\tilde{\mathcal{C}}_{\mathsf{part}(G') \setminus \{p\}}}(\mathcal{C}'_p); S, \tag{D.14}$$

where

$$\begin{aligned} S = (\nu \mathcal{C}'_q)_{q \in \mathsf{part}(G') \setminus \{p\}} \Big(&\|_{q \in \mathsf{part}(G') \setminus \{p\}} \mathcal{C}_q.\overline{\mathtt{some}}\langle \mathcal{C}'_q \rangle \\ &\| \, \mathsf{M}[\![G']\!]_{\mathcal{C}\{q \mapsto \mathcal{C}'_q\}_{q \in \mathsf{part}(G')}} \Big). \end{aligned} \tag{D.15}$$

Since we assume G is SWF, by Lemma D.2, subtype G' is also SWF. Therefore, by the induction hypothesis,

$$\begin{aligned} \mathsf{M}[\![G']\!]_{\mathcal{C}'} \vdash &\mathcal{C}'_p : \langle\!\langle G' \upharpoonright p \rangle\!\rangle, \\ &\mathcal{C}'_{p_2} : \langle\!\langle G' \upharpoonright p_2 \rangle\!\rangle, \ldots, \mathcal{C}'_{p_n} : \langle\!\langle G' \upharpoonright p_n \rangle\!\rangle; \Theta \end{aligned}$$

is a compositional typing for some Θ, where $\mathcal{C}' \triangleq \mathcal{C}\{q \mapsto \mathcal{C}'_q\}_{q \in \mathsf{part}G'}$.

W.l.o.g. we assume $n = 3$, i.e. $\mathsf{part}(G) = \{p = p_1, p_2, p_3\}$. Figure D.2 shows the typing inference of S (given in (D.15)). Note that the result of this derivation is not exactly equal to S, but they are structurally congruent, which is good enough. By applying the Non-deterministic Select Rule with the derivation of S as premise, we obtain a compositional typing for $\mathsf{M}[\![G]\!]_{\mathcal{C}}$ (cf. (D.14)):

$$\mathsf{M}[\![G]\!]_{\mathcal{C}} \vdash \mathcal{C}_p : \oplus\overline{\langle\!\langle G' \upharpoonright p \rangle\!\rangle}, \mathcal{C}_{p_2} : \&\overline{\langle\!\langle G' \upharpoonright p_2 \rangle\!\rangle}, \mathcal{C}_{p_3} : \&\overline{\langle\!\langle G' \upharpoonright p_3 \rangle\!\rangle}$$

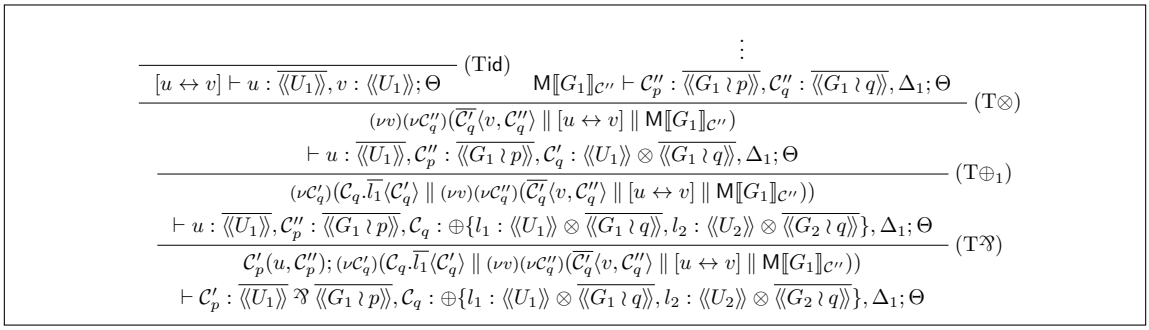


Figure D.1: Type derivation of (D.6).

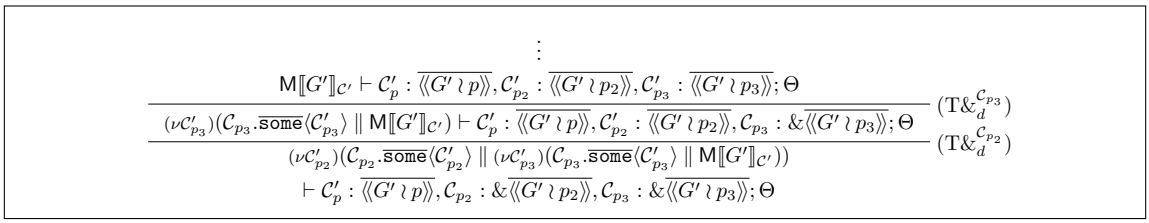


Figure D.2: Type derivation of (D.15).

□

This shows that the medium process of a simply well-formed global type induces a valid typing that coincides with the binary local types of all of the global types' participants. Does the converse hold as well? More specifically, given a valid typing of a global types' medium process, do the binary types of the participants coincide with the binary local types of the global type? It turns out we can only prove a slightly weaker statement. Let us explore why.

Suppose we are given some global type G with $\mathsf{part}(G) = \{p_1, \ldots, p_n\}$. Also, we have the following compositional typing (so it is valid and contains typings for all of G's participants), where $\mathcal{C} = \mathsf{npart}(G)$:

$$\mathsf{M}[\![G]\!]_{\mathcal{C}} \vdash \mathcal{C}_{p_1} : \overline{A_1}, \ldots, \mathcal{C}_{p_n} : \overline{A_n}; \Theta$$

What we want is that

$$\langle\!\langle G \upharpoonright p_j \rangle\!\rangle = A_j \text{ for every } j \in \{1, \ldots, n\}.$$

However, consider the case when $G = p \twoheadrightarrow q : \{l_i\langle U_i\rangle : G_i\}_{i \in I}$ (assuming w.l.o.g. that $p = p_1$ and $q = p_2$). The medium process looks as follows:

$$\mathsf{M}[\![G]\!]_{\mathcal{C}} = \mathcal{C}_p.\mathsf{case}(\mathcal{C}'_p)\,(l_i : L_i)_{i \in I}\,,$$

where each L_i is as in (D.6) and (D.7). By reverse typing we find that L_i must have been typed using the n-ary Select Rule (T$\oplus_{i \in J}$). Hence, $A_q = \&\{l_j : B_j\}_{j \in J}$ and thus

$$L_i \vdash \mathcal{C}_q : \oplus\{l_j : \overline{B_j}\}_{j \in J}, \ldots$$

The problem is that this rule only requires the selected index to be present in the selection type ($i \in J$). This means that it is not necessarily the case that $I = J$. Since the n-ary Offer Rule (used to type $\mathsf{M}[\![G]\!]_{\mathcal{C}}$ with the L_i as premises) requires the linear context across premises to be equal, and $i \in J$ for every $i \in I$, we do know that $I \subseteq J$. Now what happens if $I \subset J$? Then our desired result fails:

$$\langle\!\langle G \wr q \rangle\!\rangle \neq A_q$$

Now we are ready to prove a (slightly) weaker statement.

Theorem D.3

Let G be a global type with $\mathsf{part}(G) = \{p_1, \ldots, p_n\}$ and let $\mathcal{C} = \mathsf{npart}(G)$. Given binary types $A_1, \ldots, A_n$, if

$$\mathsf{M}[\![G]\!]_{\mathcal{C}} \vdash \mathcal{C}_{p_1} : \overline{A_1}, \ldots, \mathcal{C}_{p_n} : \overline{A_n}; \Theta$$

is a compositional typing for some persistent context Θ, then there exist $T_1, \ldots, T_n$ such that

$$\langle\!\langle G \wr p_j \sqcup T_j \rangle\!\rangle = A_j$$

for every $p_j \in \mathsf{part}(G)$.

Proof. By induction on the structure of G. However, by looking at all possible reverse typings, we find there is only one important case. For all the other cases, we can use $T_j = G \wr p_j$, i.e. $\langle\!\langle G \wr p_j \sqcup G \wr p_j \rangle\!\rangle = \langle\!\langle G \wr p_j \rangle\!\rangle = A_j$. The important case is $G = p \twoheadrightarrow q : \{l_i\langle U_i \rangle : G_i\}_{i \in I}$.

W.l.o.g. we assume $p = p_1$ and $q = p_2$. The only important participant is q, since they have a reverse typing using the n-ary Select Rule. We restate the medium process and the typing of q's channel:

$$\mathsf{M}[\![G]\!]_{\mathcal{C}} = \mathcal{C}_p.\mathsf{case}(\mathcal{C}'_p)\,(l_i : L_i)_{i \in I}$$
$$L_i \vdash \mathcal{C}_q : \oplus\{l_j : \overline{B_j}\}_{j \in J}, \ldots$$

Hence, $A_q = \&\{l_j : B_j\}_{j \in J}$. Also, as we established before, $I \subseteq J$.

By the induction hypothesis, we can find local type T'_i for every $i \in I$ such that $B_i = \overline{U_i} \parr \langle\!\langle G_i \wr q \sqcup T'_i \rangle\!\rangle$. For every $j \in J \setminus I$ we can use an arbitrary binary type U_j and arbitrary local type T'_j and let $B_j = U_j \parr \langle\!\langle T'_j \rangle\!\rangle$, since that type will never be reached by q's channel anyway. This covers all the labels in A_q (so every index of $J = I \cup (J \setminus I)$).

Now we can define T_q:

$$T_q = p?\{l_i\langle U_i \rangle : (G_i \wr q \sqcup T'_i), l_j\langle U_j \rangle : T'_j\}_{i \in I, j \in J \setminus I} \tag{D.16}$$

It is easy to see that $\langle\!\langle T_q \rangle\!\rangle = A_q$. The simple projection of G under q is:

$$G \wr q = p?\{l_i\langle U_i \rangle : (G_i \wr q)\}_{i \in I} \tag{D.17}$$

We can see from (D.16) and (D.17) that our desired result holds:

$$\langle\!\langle G \wr q \sqcup T_q \rangle\!\rangle = \langle\!\langle T_q \rangle\!\rangle = A_q$$

□

D.3.2 Merge well-formedness

Now that we have established our results for well-formedness based on the simple projection, we can focus on the more general definition based on projection with merge operations. We restate Lemma D.2 and Theorems D.1 and D.3 for merge well-formedness.

> **Lemma D.4**
>
> i) If $p \twoheadrightarrow q : \{l_i\langle U_i\rangle : G_i\}_{i\in I}$ is MWF, then for all $i \in I$, G_i is MWF too, ii) if $(G_1 \parallel G_2)$ is MWF, then G_1 and G_2 are MWF too, and iii) if $p \,\&\, G$ is MWF, then G is MWF too.

Proof. Exactly as the proof of Lemma D.2. □

Theorem D.5

Let G be a global type with $\mathsf{part}(G) = \{p_1, \ldots, p_n\}$ and let $\mathcal{C} = \mathsf{npart}(G)$. If G is MWF, then

$$\mathsf{M}[\![G]\!]_{\mathcal{C}} \vdash \mathcal{C}_{p_1} : \overline{\langle\!\langle G \upharpoonright p_1 \rangle\!\rangle}, \ldots, \mathcal{C}_{p_n} : \overline{\langle\!\langle G \upharpoonright p_n \rangle\!\rangle}; \Theta$$

is a compositional typing for some persistent context Θ.

Proof. By following the steps of the proof of [5, Thm. 11], but using Lemma D.4. The key is that the n-ary Select Rule allows an arbitrary set of options. □

Theorem D.6

Let G be a global type with $\mathsf{part}(G) = \{p_1, \ldots, p_n\}$ and let $\mathcal{C} = \mathsf{npart}(G)$. Given binary types $A_1, \ldots, A_n$, if

$$\mathsf{M}[\![G]\!]_{\mathcal{C}} \vdash \mathcal{C}_{p_1} : \overline{A_1}, \ldots, \mathcal{C}_{p_n} : \overline{A_n}; \Theta$$

is a compositional typing for some persistent context Θ, then there exist $T_1, \ldots, T_n$ such that

$$\langle\!\langle G \upharpoonright p_j \sqcup T_j \rangle\!\rangle = A_j$$

for every $p_j \in \mathsf{part}(G)$.

Proof. By following the steps of the proof of [5, Thm. 12]. The intuition is that in the $G = p \twoheadrightarrow q : \{l_i\langle U_i\rangle : G_i\}_{i\in I}$ case, the different local types across branches G_i are already merged as per Definition D.4. □